PEARL 90

Sprachreport

Version 2.0

Januar 1995

Herausgeber: GI-Fachgruppe 4.4.2 "Echtzeitprogrammierung, PEARL"

Herausgeber: GI-Fachgruppe 4.4.2
Echtzeitprogrammierung, PEARL

Sprecher:	Stellvertreter:
Dr. Hans Windauer	Dr. Peter Holleczek
Werum GmbH	Universität Erlangen-Nürnberg Regionales Rechenzentrum
Erbstorfer Landstr. 14 21337 Lüneburg Tel. 04131/8900-66 Fax 04131/8900-20	Martensstraße 1 91058 Erlangen Tel. 09131/85-7817 Fax 09131/302941

Innerhalb der Fachgruppe ist der Arbeitskreis 5 für die Pflege des Sprachreports zuständig. Leiter dieses Arbeitskreises ist

Priv.-Doz. Dr.-Ing. Georg Thiele

Universität Bremen
Fachbereich 1 (Physik/Elektrotechnik)
Postfach 33 04 40
28334 Bremen
Tel. 0421/218-2442
Fax 0421/218-4707

Inhaltsverzeichnis

1. Einführung

PEARL steht für **P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage; es ist eine höhere Programmiersprache, die eine komfortable, sichere und weitgehend rechnerunabhängige Programmierung von Multitasking- und Echtzeit-Aufgaben erlaubt. PEARL wurde in verschiedenen Ausbaustufen vom DIN genormt:

- DIN 66253, Teil 1, Basic PEARL, 1981
  (Subset von Full PEARL)
- DIN 66253, Teil 2, Full PEARL, 1982
- DIN 66253, Teil 3, Mehrrechner PEARL, 1988

Auf Basis der Erfahrungen aus Hunderten von PEARL-Projekten erarbeitete in den Jahren 1989 und 1990 ein Gremium aus PEARL-Anwendern und -Implementatoren die Definition von PEARL 90 (vgl. Stieger, K.: PEARL 90 - Die Weiterentwicklung von PEARL. In: Informatik Fachberichte 231, PEARL '89 Workshop über Realzeit-systeme, Springer 1989).

PEARL 90 entspricht Full PEARL, wobei einige, in der Praxis nicht benötigte Sprach-elemente entfallen. Andererseits enthält PEARL 90 einige fortschrittliche Erweiterun-gen.

Der vorliegende Sprachreport beschreibt den Sprachumfang von PEARL 90, wobei er zunächst die wesentlichen Merkmale von PEARL 90 skizziert. Anschließend werden die Sprachelemente genau definiert.

Der Anhang enthält eine Aufstellung aller Datentypen und ihrer Verwendungs-möglichkeiten, eine Beschreibung der verfügbaren vordefinierten Funktionen, eine vollständige Darstellung der Syntax, da in den einzelnen Abschnitten aus didaktischen Gründen nicht immer alle syntaktischen Möglichkeiten beschrieben werden, sowie eine Liste aller Schlüsselwörter.

2. Wesentliche Merkmale von PEARL

2.1 Multitasking-Eigenschaften

Ein DV-Programm zur on-line-Steuerung oder -Auswertung eines technischen Prozesses muß möglichst schnell auf spontane Meldungen des Prozesses oder zeitliche Ereignisse reagieren können. Deshalb genügt es meist nicht, die einzelnen Programmschritte sequentiell, d.h. in zeitlich unveränderlicher Reihenfolge anzuordnen und durchzuführen. Vielmehr muß die mehr oder weniger komplexe Automationsaufgabe in problemgerechte Teilaufgaben mit unterschiedlicher Dringlichkeit zerlegt und die Programmstruktur dieser Aufgabenstruktur angepaßt werden. Dabei entstehen selbständige Programmteile für Teilaufgaben, die zeitlich sequentiell innerhalb anderer Aufgaben bearbeitet werden können (z.B. Prozeduren); es entstehen aber auch selbständige Programmelemente für Teilaufgaben, die aufgrund eines zeitlich nicht unbedingt festgelegten Anstoßes (z.B. einer Störmeldung aus dem Prozeß) sofort zeitlich parallel zu allen anderen Aufgaben bearbeitet werden müssen. Die Ausführung eines solchen Programmelements wird Task genannt; zur Festlegung ihrer Dringlichkeit können Tasks mit Prioritäten versehen werden.

Für die Vereinbarung und das Zusammenspiel von Tasks untereinander und mit dem technischen Prozeß bietet PEARL folgende Möglichkeiten:

- Vereinbarung von Tasks, z.B.

  Nachschub: **TASK PRIORITY** 2 ;
  Taskkörper (Vereinbarungen, Anweisungen)
  END ;

- Start (Aktivierung), z.B.

  ACTIVATE Nachschub ;

- Beenden, z.B.

  TERMINATE Drucken ;

- Anhalten, z.B.

  SUSPEND Statistik ;

- Fortsetzen, z.B.

  CONTINUE Statistik ;

- Verzögern, z.B.

  AFTER 5 SEC **RESUME** ;

Entsprechend den Anforderungen von Automationsaufgaben können manche dieser Anweisungen für die (wiederholte) Ausführung eingeplant werden, z.B. für den Fall des Eintritts einer Uhrzeit, den Ablauf einer Zeitdauer oder den Eintritt einer Meldung:

WHEN fertig **ACTIVATE** Nachschub ;

(Bedeutung: Jedes Mal, wenn der Interrupt fertig auftritt, ist die Task Nachschub zu aktivieren)

Einplanungen können auch den zeitlich periodischen Start vorsehen:

AT 12:0:0 **ALL** 1 SEC **UNTIL** 12:15:0 **ACTIVATE** Messen ;

Sofern nicht entsprechende Maßnahmen dies verhindern, führen verschiedene Tasks ihre Anweisungen unabhängig voneinander aus. Mitunter ist jedoch eine Synchronisierung zweier oder mehrerer Tasks erforderlich, z.B. wenn eine Task Daten erzeugt und in einen Puffer ablegt. Hier darf der Erzeuger nicht schneller arbeiten als der Verbraucher. Komplexere Synchronisationsprobleme treten auf, wenn z.B. eine Task exklusiv (weil schreibend) auf eine Datei zugreifen muß, andere jedoch untereinander simultan (weil lesend) zugreifen können. Zur Lösung solcher Synchronisationsprobleme enthält PEARL die Datentypen SEMA und BOLT mit zugehörigen Anweisungen, wie z.B.

- **REQUEST** Foerderstrecke ;
- **RELEASE** Kommunikationspuffer ;

Wesentlich ist, daß diese Multitasking-Anweisungen *rechnerunabhängig* sind, d.h. PEARL-Programme können ohne Änderung z.B. auf iRMX-, OS/2-, Unix- oder VAX/VMS-Systemen ablaufen.

Hinzu kommt der große Vorteil, daß der PEARL-Programmierer sein Multitasking-Problem *problemorientiert* mit hohem Komfort programmieren kann, ohne sich in die Eigenheiten der verschiedenen Betriebssysteme vertiefen zu müssen - z.B. in die Handhabung von Fork- und Message-queue-Mechanismen bei Unix. Die Umsetzung der problemorientierten Multitasking-Anweisungen von PEARL in die Mechanismen der Multitasking-Betriebssysteme übernimmt der PEARL-Compiler.

2.2 Ein- und Ausgabe-Möglichkeiten

Die Übertragung von Daten zu bzw. von Geräten der Standardperipherie (Drucker, Lochkartenleser, Platte usw.) oder Prozeßperipherie (Meßwertgeber, Stellglieder usw.) sowie die Verwaltung von Dateien erfolgen in PEARL mit Hilfe rechnerunabhängiger Anweisungen.

Geräte und Dateien werden unter dem Begriff Datenstation zusammengefaßt. Im wesentlichen werden zwei Arten der Datenübertragung unterschieden:

- Die Übertragung der Daten ohne Formatsteuerung, d.h. ohne Umwandlung in (oder an) eine(r) externe(n) Darstellung:

  Diese Art der Datenübertragung wird für den Dateiverkehr, der sequentielle und direkte Zugriffe erlaubt, sowie für die Übertragung von Prozeßdaten eingesetzt.

  Beispiele:

  ```
  READ   Stammsatz FROM Stammdatei BY POS (10) ;
  WRITE  Datensatz TO Logbuch;
  TAKE   Meßwert FROM TemperaturSensor;
  SEND   Ein TO Motor ;
  ```

- Die Übertragung der Daten mit Formatsteuerung, d.h mit Umwandlung zwischen internem Format und der externen Darstellung mit den auf der Datenstation zur Verfügung stehenden Möglichkeiten:

  Damit ist z.B. die Darstellung in den Zeichen des Zeichensatzes der Datenstation gemeint.

  Beispiele:

  ```
  PUT   Ereignis TO Drucker BY F(5) ;
  GET   Quittung FROM Terminal;
  ```

Die Namen der Datenstationen sind frei wählbar. Dies wird durch die Aufteilung eines PEARL-Programms in rechnerabhängige und weitgehend rechnerunabhängige Teile erreicht.

Zur Ansprache spezieller Geräte bietet der Compiler ein Treiber-Interface, an das der PEARL-Programmierer selbst Gerätetreiber anschließen kann.

2.3 Programmstruktur

Programmsysteme zur Lösung komplexer Automationsaufgaben sollten modular aufgebaut werden. PEARL kommt dieser Forderung dadurch entgegen, daß ein PEARL-Programm aus einem oder mehreren getrennt übersetzbaren Moduln besteht.

Bezüge zwischen Moduln sind mittels sogenannter globaler Objekte (z.B. Variablen, Prozeduren, Tasks) möglich.

Damit die Anweisungen für die Datenübertragung sowie für die Einplanung von Reaktionen auf Meldungen des technischen Prozesses (Interrupts) oder der Rechnerhardware (Signale) rechnerunabhängig programmiert werden können, gliedert sich ein Modul normalerweise in einen Systemteil und einen Problemteil.

Im Systemteil wird die eingesetzte Hardware-Konfiguration beschrieben. Insbesondere können hier den Geräten und ihren Verbindungen, den Interrupts und Signalen, frei gewählte Namen zugeordnet werden. So besagt das folgende Beispiel, daß ein Ventil an dem Anschlußpunkt 3 eines Digitalausgabewerks angeschlossen ist, das den rechnerspezifischen Systemnamen DIGOUT (1) trägt. Das Vertil, d.h. der Anschlußpunkt 3, soll den frei wählbaren rechnerunabhängigen Benutzernamen "Ventil" bekommen:

```
Ventil : DIGOUT (1) * 3 ;
```

Unter Verwendung der im Systemteil eingeführten Benutzernamen wird im Problemteil der eigentliche Algorithmus zur Lösung der Automationsaufgabe rechnerunabhängig programmiert, z.B.:

```
SEND  auf  TO  Ventil ;
```

Zur Strukturierung der Algorithmen stehen (benannte) Blöcke, Prozeduren und Tasks (parallele Aktivitäten) zur Verfügung.

2.4 Datentypen

Standardmäßig stehen in PEARL folgende Datentypen zur Verfügung:

- **FIXED** und **FLOAT** mit angebbarer Genauigkeit
- **BIT** - und **CHARACTER** -Strings mit angebbarer Länge
- **CLOCK** und **DURATION** für Uhrzeiten und Zeitdauern
- Referenzen (**REF**) für indirekte Adressierung
- Geräte und Dateien (**DATION**) für Standard- und Prozeß-Ein-/Ausgabe
- Interrupts (**INTERRUPT**) für externe Unterbrechungen
- Signale (**SIGNAL**) für interne Ausnahmesituationen
- Semaphoren (**SEMA**) und Boltvariablen (**BOLT**) zur Koordination des Zugriffs von Tasks auf gemeinsam benutzte Objekte

Der Benutzer kann daraus selbst komplexere Datenstrukturen wie Felder, hierarchische Strukturen (STRUCT) und Listen aufbauen; mittels der TYPE -Definition können diese auch als neue, problemorientierte Datentypen vereinbart werden. Außerdem erlaubt PEARL die Einführung neuer, problemorientierter Operatoren für beliebige Datenstrukturen mittels der OPERATOR -Definition.

2.5 Kontrollstrukturen

Die folgenden Kontrollstrukturen stehen zur Verfügung:

- Bedingte Anweisung

IF Ausdruck
THEN Anweisung ···
ELSE Anweisung ···
FIN ;

- Anweisungsauswahl_1

CASE Ausdruck
ALT Anweisung ···
ALT Anweisung ···
...
OUT Anweisung ···
FIN ;

- Anweisungsauswahl_2

CASE Case_Index
ALT (Case_Liste) Anweisung···
ALT (Case_Liste) Anweisung ···
...
OUT Anweisung ···
FIN ;

- Schleifen

FOR Laufvariable **FROM** Anfang **BY** Schrittweite **TO** Ende
REPEAT
Anweisung ···
END ;

WHILE Bedingung
REPEAT
Anweisung ···
END ;

- Exit-Anweisung

EXIT Block ;

3. Regeln zum Aufbau von PEARL-Sprachformen

Ein PEARL-Programm kann formatfrei geschrieben werden; es muß insbesondere nicht darauf geachtet werden, daß eine Anweisung in einer bestimmten Spalte beginnt.

Alle Elemente eines PEARL-Programms werden aus Zeichen des folgenden Zeichensatzes erstellt. Allerdings dürfen Zeichenkettenkonstanten und Kommentare jedes Zeichen enthalten, das ISO 646 und nationale Varianten von ISO 646 zulassen.

3.1 Zeichensatz

Der Zeichensatz von PEARL ist eine Teilmenge des ISO 7-Bit Zeichensatzes (ISO 646). Er enthält

- die Großbuchstaben A bis Z
- die Kleinbuchstaben a bis z
- die Ziffern 0 bis 9 und
- die Sonderzeichen

_	Unterstrich
	Zwischenraum (zur Verdeutlichung wird manchmal auch der Unterstrich _ benutzt)
!	Ausrufungszeichen (Beginn des Zeilenkommentars)
'	Apostroph
(	runde Klammer auf
)	runde Klammer zu
,	Komma
.	Punkt
;	Semikolon
:	Doppelpunkt
+	Pluszeichen (z.B. für Addition, Vorzeichen)
-	Minuszeichen (z.B. für Subtraktion, Vorzeichen)
*	Stern (z.B. für Multiplikation)
/	Schrägstrich (z.B. für Division)
=	Gleichheitszeichen (z.B. für Zuweisung)
<	Kleiner-Zeichen
>	Größer-Zeichen
[	Strukturklammerungszeichen
]	Strukturklammerungszeichen
\	Rückschrägstrich (für Steuerzeichen in Zeichenketten)

Die folgenden Zeichenkombinationen werden jeweils als eine Einheit (zusammengesetztes Symbol) interpretiert:

**	Exponentiationssymbol
/*	Kommentar-Beginn
*/	Kommentar-Ende
//	Symbol für ganzzahlige Division
==	Gleich-Symbol
/=	Ungleich Symbol
<=	Kleiner-Gleich-Symbol
>=	Größer-Gleich-Symbol
<>	Cyclic-Shift-Symbol
><	Verkettungssymbol
'\	Einleitung einer Steuerzeichensequenz in Zeichenkettenkonstanten
\'	Ende einer Steuerzeichensequenz in Zeichenkettenkonstanten

Falls auf dem Gerät für die Programmniederschrift nicht alle Symbole zur Verfügung stehen, können folgende Zeichenfolgen alternativ benutzt werden:

LT	für <
GT	für >
EQ	für ==
NE	für /=
LE	für <=
GE	für >=
CSHIFT	für <>
CAT	für ><
(/	für [
/)	für]

3.2 Grundelemente

Ein PEARL-Programm wird aus folgenden Grundelementen aufgebaut:

- Bezeichner
- Konstanten
- Trennzeichen (das sind Sonderzeichen und zusammengesetzte Symbole) und
- Kommentare

Auf die Zeichenfolge für Bezeichner, und Konstanten müssen Trennzeichen oder Kommentare folgen.

3.2.1 Bezeichner

Bezeichner werden zur Bildung von Namen für Objekte (z.B. Zahlenvariablen, Prozeduren) benutzt. Sie bestehen aus einer Folge von Buchstaben (groß und klein), dem Unterstrich und/oder Ziffern; diese Folge muß mit einem Buchstaben beginnen. PEARL unterscheidet bei Bezeichnern zwischen großen und kleinen Buchstaben, d.h. ventil, VENTIL und Ventil bezeichnen verschiedene Objekte.

Beispiele: Zaehler_1, DISPO, warten

Einige Wörter haben an festgelegten Stellen im PEARL-Programm eine spezifische Bedeutung; diese Wörter werden Schlüsselwörter genannt. Z.B. sind die Wörter BIT oder GOTO solche Schlüsselwörter. Der Anhang enthält eine Liste aller Schlüsselwörter. Sie dürfen nicht als Bezeichner benutzt werden und müssen immer in Großbuchstaben geschrieben sein.

3.2.2 Konstanten

Konstanten sind ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Uhrzeiten und Zeitdauern. Sie werden zusammen mit den entsprechenden Variablen in Kapitel 5 beschrieben.

3.2.3 Kommentare

Kommentare dienen zur Erläuterung des Programms und sind ohne Bedeutung für den Programmablauf. Es gibt zwei Arten von Kommentaren: Die eine Art kann sich über mehrere Zeilen erstrecken und wird durch die zusammengesetzten Symbole “/*“ und “*/“ eingeklammert. Innerhalb dieser Klammern können beliebige Zeichen erscheinen mit Ausnahme des zusammengesetzten Symbols “*/“ für Kommentar-ende.

Die andere Art, der Zeilenkommentar, beginnt mit einem Ausrufungszeichen “!“ und wird durch das Zeilenende beendet.

Kommentare dürfen überall eingefügt werden, wo Leerzeichen zulässig sind.

Beispiele:

```
/*  Dieser Kommentar wird durch das Zeilenende
    nicht beendet  */
!   Dieser Kommentar ist auf 1 Zeile begrenzt
```

3.3 Aufbau von Sprachformen

In den nachfolgenden Kapiteln werden die in PEARL zulässigen Sprachformen beschrieben. Damit diese Beschreibungen genau und möglichst kompakt sind, werden über die verbale Formulierung hinaus einige formale Möglichkeiten benötigt:

Jede Sprachform hat einen Namen, unter dem sie mit Hilfe des (Meta-)Symbols “::=“ definiert wird:

```
Name-der-Sprachform ::=
        Definition-der-Sprachform
```

Beispiel:

```
Großbuchstabe ::=
        A oder B oder . . . oder Z

Ziffer ::=
        0 oder 1 oder . . . oder 9
```

Wie dieses Beispiel zeigt, kann die Definition einer Sprachform Elemente enthalten, die beim Aufbau der Sprachform alternativ zueinander angegeben werden. Zur Abkürzung werden die alternativ möglichen Angaben künftig durch das Symbol “|“ getrennt:

Beispiel:

```
Buchstabe ::=
        A | B | . . . | Z | a | b | . . . | z

Ziffer ::=
        0 | 1 | 2 | . . . | 9
```

Soll ein Element beliebig oft, aber mindestens einmal, auftreten dürfen, wird es oben mit drei Punkten versehen.

Beispiel:

```
einfache-ganze-Zahl ::=
        Ziffer ···
```

Um auszudrücken, daß ein Element beim Aufbau der Sprachform fehlen darf, wird es in eckigen Klammern “[“...“]“ angegeben (Im Zweifelsfall werden die Strukturklammerungszeichen fett gedruckt).

Beispiel:

Bezeichner ::=
    Buchstabe [{ Buchstabe | Ziffer | _ } ···]

Hier wurden bereits zwei weitere Regeln benutzt: Die Definition einer Sprachform darf wieder Namen von Sprachformen enthalten; außerdem werden die geschweiften und eckigen Klammern auch dazu benutzt, Elemente zu neuen Elementen zusammenzufassen. So ist das letzte Beispiel dem folgenden äquivalent:

Beispiel:

Bezeichner ::=
    Buchstabe [Buchstabe | Ziffer | _] ···

Zur Beschreibung von Listen, deren Elemente durch ein bestimmtes Symbol getrennt sind, werden bei der Definition der entsprechenden Sprachform das Listenelement und das Trennsymbol in der Form

Listenelement [Trennsymbol Listenelement] ···

angegeben.

Beispiel:

Bezeichnerliste ::=
    Bezeichner [, Bezeichner] ···

Zum besseren Verständnis oder zur genaueren Beschreibung der Definition einer Sprachform werden häufig Elemente mit einem erläuternden oder einschränkenden Kommentar versehen, der von dem Element durch das Symbol “§“ getrennt wird.

Beispiel:

Geräteliste ::=
    Bezeichner§Gerät [, Bezeichner§Gerät] ···

4. Programmstruktur

4.1 Moduln

Ein PEARL-Programm setzt sich aus einem oder mehreren Teilen, sogenannten Moduln, zusammen, die unabhängig übersetzt werden. Jeder Modul besteht aus einem Systemteil und/oder einem Problemteil.

Die allgemeine Form eines PEARL-Programms lautet:

```
PEARL-Programm ::=
        Modul ···

Modul ::=
        MODULE   [ (Bezeichner§des-Moduls) ] ;
                { Systemteil  [ Problemteil ] }  |  Problemteil
        MODEND ;

Systemteil ::=
        SYSTEM ; [ Vereinbarung ··· ]

Problemteil ::=
        PROBLEM ; [ Vereinbarung ··· ]
```

Im Systemteil werden die Verbindungen des projektierten Rechners mit den Elementen des technischen Prozesses (Meßwertgeber, Stellglieder, usw.) und der Standardperipherie (Tastatur, Monitor, Drucker, Platten, Bänder usw.) beschrieben. Dabei kann der Programmierer den Eingängen zum Interruptwerk und den in E/A-Anweisungen (im Problemteil) angesprochenen Peripherieelementen frei wählbare Namen zuordnen, um sich im Problemteil auf diese (rechnerunabhängigen) Namen beziehen zu können.

Im Problemteil wird der Algorithmus zur Lösung der gestellten Automationsaufgabe beschrieben. Hierzu vereinbart der Programmierer folgende Objekte:

- Variablen und Konstanten für ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Zeitdauern, Uhrzeiten, Referenzen
- Sprungmarken
- Prozeduren für mehrfach auftretende Teilaufgaben
- Tasks für die zeitliche parallele Abarbeitung von Aufgaben
- Blöcke für die Strukturierung von Prozeduren und Tasks

- Interrupts
- Signale
- Synchronisiervariablen (Sema- und Bolt-Variablen)   sowie
- Datenstationen und Formate für die Ein/Ausgabe

Die erforderlichen Anweisungen werden in den Prozeduren und Tasks zusammen mit weiteren "lokalen" Vereinbarungen, die nur dort benötigt werden, angegeben. Dabei gilt allgemein, daß Objekte erst dann (in Anweisungen) benutzt werden dürfen, wenn sie vereinbart worden sind.

Beispiel:

```
MODULE ;

SYSTEM ;
    Beschreibung der Verbindungen und Einführung von
    Namen für die Peripherieelemente

PROBLEM ;
    Vereinbarung von Konstanten und Variablen
    Vereinbarung von Interrupts
    Vereinbarung von Datenstationen

    Vereinbarung einer Task
        Vereinbarung lokaler Konstanten und Variablen
        Anweisungen

    Vereinbarung einer Prozedur
        Vereinbarung lokaler Konstanten und Variablen
        Vereinbarung lokaler Prozeduren
        Anweisungen

    ...

MODEND ;
```

Objekte werden auf Modulebene, d.h. außerhalb von Prozeduren und Tasks, oder in Prozeduren oder in Tasks vereinbart. Auf Modulebene vereinbarte Objekte sind im ganzen Modul bekannt und können von jeder Task und Prozedur des Moduls unter Angabe des Bezeichners benutzt oder gegebenenfalls geändert werden. Ein in einer Task oder Prozedur vereinbartes Objekt ist nur in der betreffenden Task oder Prozedur bekannt und kann nur dort benutzt oder gegebenenfalls geändert werden.

4.2 Vereinbarungen

Die Vereinbarung von Objekten erfolgt durch *Deklaration* oder durch *Spezifikation*.

4.2.1 Deklaration (DCL)

Die Deklaration dient dazu, ein Objekt und seinen Namen einzuführen, d.h. bei Auswertung der Deklaration wird Speicherplatz für das Objekt angelegt, und es kann ab diesem Zeitpunkt unter dem in der Deklaration angegebenen Namen darauf zugegriffen werden.

Auf Modulebene oder in einer Prozedur oder in einer Task darf ein Objekt nur einmal deklariert werden. Ist ein Bezeichner X sowohl auf Modulebene als auch in einer Prozedur oder Task als Objekt deklariert, so sind damit zwei verschiedene Objekte eingeführt: In der betreffenden Prozedur oder Task bezieht man sich unter dem Bezeichner X auf das dort (lokal) deklarierte Objekt, außerhalb der Prozedur oder Task auf das auf Modulebene deklarierte Objekt (siehe auch Punkt 4.3, Blockstruktur).

Beispiel:

```
PROBLEM ;
        DECLARE  x  FLOAT;  ! 1. Deklaration auf Modulebene
        DECLARE  x  FIXED ;  ! 2. Deklaration auf Modulebene (falsch)

P :     PROCEDURE ;
        DECLARE  x  FIXED ;  ! Deklaration in der Prozedur P (zulässig)
        ...
        x := 3 ;             ! Zuweisung an die lokale Variable x
        ...
        END ; ! P

T :     TASK ;
        ...
        x := 5 ;             ! Zuweisung an die auf Modulebene deklarierte Variable x
        ...
        END ; ! T

...
```

Die verschiedenen Deklarationsformen werden in den nachfolgenden Kapiteln bei den einzelnen Objekten behandelt.

Innerhalb von Prozeduren und Tasks dürfen

- Tasks
- Interrupts
- Signale
- Synchronisiervariablen
- Datenstationen sowie
- Formate

nicht vereinbart werden.

Die folgende Tabelle zeigt, welche Objekte wo vereinbart werden dürfen:

	Vereinbarung möglich auf/in			
Objekt	Modulebene	Task	Prozedur	Block
Variablen, Konstanten	x	x	x	x
Sprungmarken	-	x	x	x
Prozeduren	x	x	x	-
Task	x	-	-	-
Block	-	x	x	x
Interrupt	x	-	-	-
Signal	x	-	-	-
Sema-, Bolt-Variablen	x	-	-	-
Datenstationen, Formate	x	-	-	-
Typen	x	x	x	x

4.2.2 Spezifikation (SPC) und Identifizierung (SPC IDENT)

Mit einer Spezifikation bezieht man sich auf ein bereits vereinbartes (deklariertes) Objekt. Dies ist sinnvoll für Objekte, die in einem Modul vereinbart und in anderen Moduln benutzt werden sollen (siehe 4.4, Bezüge zwischen Moduln), aber auch generell zur Einführung zusätzlicher Namen für bereits vereinbarte Objekte.

Beispiel:

> Das Objekt x vom Typ FIXED (15) soll xx als 2. Namen erhalten - oder anders formuliert: Das Objekt x soll mit dem Namen xx *identifiziert* werden:

```
PROBLEM ;
    ...
    DECLARE  x  FIXED ;
    ...
    SPECIFY  xx  FIXED  IDENT  (x) ;
    ...
    xx := 7 ;         ! Zuweisung an das Objekt x
```

Allgemein lautet die Form der Identifikation wie folgt:

Identifikation ::=
    { **SPECIFY** | **SPC** } Bezeichner [Zuweisungsschutz] Typ Identifikationsattribut ;

Identifikationsattribut ::=
    IDENT (Name§Objekt)

Der angegebene Typ muß dem Typ des benannten Objekts entsprechen. Genaueres wird bei der Vorstellung der einzelnen Objekte definiert.

4.3 Blockstruktur, Gültigkeit von Objekten

Blöcke werden dazu benutzt, Task- oder Prozedurkörper zu strukturieren und den Gültigkeitsbereich und die Lebensdauer von PEARL-Objekten zu beeinflussen. Ein Block ist eine Zusammenfassung von Vereinbarungen und Anweisungen:

```
Block ::=
        BEGIN
        [ Vereinbarung ··· ]
        [ Anweisung ··· ]
        END   [ Bezeichner§Block ] ;
```

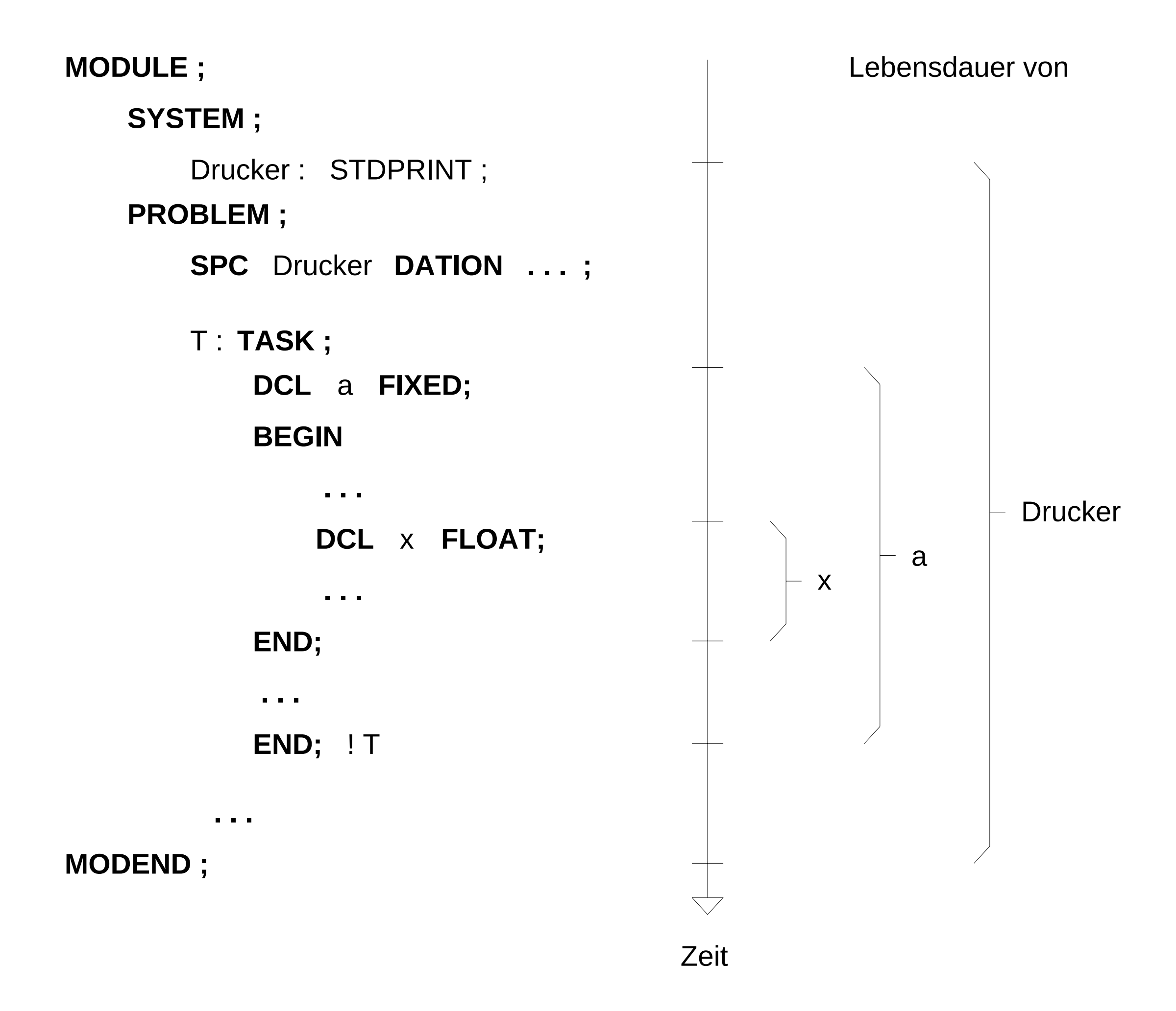
Blöcke gelten als Anweisungen und dürfen deshalb nur innerhalb von Tasks und Prozeduren auftreten, dort allerdings auch ineinander geschachtelt. Der Eintritt in einen Block erfolgt mit der Ausführung von BEGIN. Verlassen wird ein Block durch die Anweisung des zugehörigen END oder durch eine Verzweigung zu einer Anweisung außerhalb des Blocks, z.B. durch die Exit-Anweisung (vgl. 7.5). Sprünge in einen Block hinein sind nicht erlaubt.

Innerhalb von Blöcken dürfen keine Prozeduren definiert werden!

Den in einem Block vereinbarten ("lokalen") Objekten wird erst bei Eintritt in den Block Speicherplatz zugewiesen; dieser wird bei Verlassen des Blocks wieder aufgegeben. Wie Tasks, Prozeduren und Wiederholungen können Blöcke also Objekte dynamisch einführen und entfernen und bieten somit die Möglichkeit, den zur Verfügung stehenden Speicherplatz mehrfach zu nutzen.

Deshalb müssen einige Regeln für die Lebensdauer und den Gültigkeitsbereich dieser Objekte getroffen werden:

Die Lebensdauer eines Objekts ist die (Ausführungs-) Zeit zwischen der Auswertung seiner Deklaration und der Ausführung des Endes des Blocks (oder der Wiederholung oder der Prozedur oder der Task oder des Moduls), in dem (oder der) die Deklaration erfolgt.



Als Gültigkeitsbereich eines Objekts werden alle Teile des Programms bezeichnet, in denen das Objekt benutzt werden kann. Folgende Regeln sind zu beachten:

- Ein auf Modulebene vereinbartes Objekt ist auf Modulebene und in allen Tasks und Prozeduren dieses Moduls benutzbar (siehe jedoch 4.4), auch in allen eingeschachtelten Prozeduren, Blöcken und Wiederholungen mit folgender Ausnahme: Der Gültigkeitsbereich wird eingeschränkt, wenn in einer der Tasks oder Prozeduren ein anderes Objekt unter demselben Namen deklariert wird.

- Ein in einer Task, Prozedur, Wiederholung oder einem Block vereinbartes Objekt ist in dieser Task, Prozedur, Wiederholung oder diesem Block und allen darin eingeschachtelten Prozeduren, Wiederholungen und Blöcken benutzbar mit folgender Ausnahme: Der Gültigkeitsbereich wird eingeschränkt, wenn in einer der eingeschachtelten Prozeduren, Wiederholungen oder Blöcke ein anderes Objekt unter demselben Namen deklariert wird.

Beispiel:

Gültigkeitsbereich von

```
PROBLEM ;
    DCL  x  FIXED ;                              ┐
    . . .                                        │
    SPC xx  FIXED IDENT( x ) ;                   │                         ┐
    T:  TASK ;                                   │                         │
        . . .                                    ├ x FIXED                 │
        END ;  ! T                               │                         │
    P:  PROC ;                                   │              ┐          │
        DCL  y  FIXED ;                          │              │          │
        x := 2 ;                                 ┘              │          │
        BEGIN                                                   │          │
          DCL  x  FLOAT ;                ┐                      │          │
          x := 3 ;                       ├ x FLOAT              │          │
          . . .                          │                      │          ├ xx FIXED
          BEGIN                          ┘                      ├ y FIXED  │
            DCL  x  DUR ;                      ┐                │          │
            . . .                              ├ x DUR          │          │
          END ;                                ┘                │          │
                                         ┐                      │          │
          . . .                          ├ x FLOAT              │          │
        END ;                            ┘                      │          │
        y := x ;    ! y = 2                  ┐                  │          │
        . . .                                ├ x FIXED          │          │
        END ;  ! P                           │                  ┘          │
    . . .                                    │                             │
MODEND ;                                     ┘                             ┘
```

Blöcke können nach dem END mit Bezeichnern versehen werden, damit geschachtelte Blöcke mittels der Exit-Anweisung (vgl. 7.5) gezielt verlassen werden können.

4.4 Bezüge zwischen Moduln

Besteht ein PEARL-Programm aus mehreren Moduln, kann es erforderlich sein, in einem Modul vereinbarte und dort Speicherplatz belegende Objekte (Daten, Prozeduren etc.) auch in anderen Moduln zu benutzen. Hierfür werden diese (globalen) Objekte in dem Modul, in dem sie Speicherplatz belegen sollen, auf Modulebene mit dem Global-Attribut deklariert und in den anderen Moduln mit dem Global-Attribut auf Modulebene spezifiziert. Auf diese Art und Weise kann also der Gültigkeitsbereich von Objekten, die auf Modulebene deklariert sind, erweitert werden.

Beispiel:

```
MODULE ( a ) ;
PROBLEM ;
    . . .
    DCL x  FIXED  GLOBAL ;
    . . .
    x := 2 ;
    . . .
MODEND ;
```

```
MODULE ( b ) ;
PROBLEM ;
    . . .
    SPC  x   FIXED  GLOBAL  ( a ) ;
    . . .
    x := 3 ;
    . . .
MODEND ;
```

Alle im Systemteil eines bestimmten Moduls aufgeführten Datenstationen, Interrupts und Signale gelten als (implizit mit dem Global-Attribut) deklariert. Deshalb werden sie in den Problemteilen des Programms nur noch spezifiziert; dabei kann das Global-Attribut im Problemteil desselben Moduls entfallen, in allen anderen Moduln muß es angegeben werden.

Die allgemeine Form des Global-Attributs lautet:

```
Global-Attribut ::=
        GLOBAL   [ (Bezeichner§eines-Moduls) ]
```

Der gegebenenfalls angegebene Bezeichner des Moduls hat nur dokumentarischen Wert.

Bei der Spezifikation eines Objekts müssen alle Attribute aus seiner Deklaration mit Ausnahme einer angegebenen Priorität, Genauigkeit oder Länge übernommen werden. In letzterem Ausnahmefall wird für den Bereich des Programms, in dem die Spezifikation gilt, die in der entsprechenden Längenvereinbarung (vgl. 5.6) definierte Genauigkeit oder Länge eingesetzt.

Beispiel:

```
MODULE ;
PROBLEM ;
    T :  TASK  PRIO  3     GLOBAL ;
            ! Taskkörper
         END ; ! T

    P :  PROC  ( A (,) FIXED IDENT   )
                             GLOBAL ;
            ! Prozedurkörper
         END ; ! P
     . . .
MODEND ;
```

```
MODULE ;
PROBLEM ;
    SPC  T   TASK              GLOBAL ,
         P   ENTRY ( (,) FIXED IDENT  )
                               GLOBAL ;
    INIT :  TASK ;
            DCL  TAB (10, 20)   FIXED ;
             . . .
            CALL P  ( TAB ) ;
             . . .
            ACTIVATE T ;
        END ; ! INIT
     . . .
MODEND ;
```

Die verschiedenen Formen der Spezifikation globaler Objekte werden bei der Vorstellung der Objekte definiert.

4.5 Ausführung eines Programms

Nach dem Laden eines PEARL-Programms startet das PEARL-Laufzeitsystem automatisch alle Tasks, die mit dem Attribut MAIN gekennzeichnet sind, gemäß ihrer Priorität. Alle mit MAIN versehenen Tasks müssen in einem Modul deklariert sein.

Beispiel:

```
MODULE   (Main) ;
SYSTEM ; ...
PROBLEM ;
Start :      TASK   MAIN ;
             ! Aktivierung und Einplanungen anderer Tasks
             END ; ! Start

Messen :     TASK   PRIO   1 ;
             ! Taskkörper
             END ; ! Messen
...
MODEND ;
```

Nach dem Laden wird zuerst die Task Start gestartet.

5. Variablen und Konstanten

5.1 Vereinbarung von Variablen und Konstanten

Bei seiner Ausführung benutzt und verändert ein PEARL-Programm u.a. ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Uhrzeiten und Zeitdauern. Diese Daten treten in Form von Konstanten oder als Werte von Variablen auf. Konstanten identifizieren sich durch ihre Schreibweise und behalten ihren Wert während des gesamten Programmablaufs. Variablen bezeichnen Daten (ihre Werte), die sich während des Programmablaufs ändern können.

Der Wertebereich einer Variablen ist im allgemeinen auf eine Art von Daten, z.B. Bitketten, beschränkt, die den Typ der Variablen bestimmt. (Bitketten-Variablen haben nur Bitketten als Werte.) Dieser Typ muß bei der Deklaration oder Spezifikation einer Variablen zusammen mit ihrem Bezeichner festgelegt werden.

Beispiel:

```
DECLARE  Schalter  BIT ;     /* Deklaration einer Variablen Schalter vom Typ Bitkette */
                             /* der Länge 1 */

SPECIFY  Status  BIT (16)  GLOBAL ;     /* Spezifikation einer globalen Variablen
                                           Status vom Typ Bitkette der Länge 16 */
```

Eine Variable bezeichnet *ein* Datenelement, z.B. *eine* ganze Zahl, *eine* Bitkette usw.; im folgenden werden zunächst solche skalaren Variablen behandelt. Die Möglichkeiten, skalare Variablen zu Feldern und Strukturen zusammenzufassen, sind in 5.10 und 5.11 beschrieben.

Bei der Deklaration einer Variablen muß ihr Typ in einem Typ-Attribut angegeben werden; sollen verschiedene Variablen mit gleichem Typ-Attribut vereinbart werden, so kann dies in Form einer Liste in einer Deklaration erfolgen; z.B. werden durch

```
DECLARE  (x, y, z)  FLOAT ;
```

die drei Variablen x, y und z mit dem Typ-Attribut FLOAT vereinbart. Der einfacheren Schreibweise wegen dürfen verschiedene Vereinbarungen in einer Deklaration formuliert werden, indem sie durch Kommata getrennt werden:

```
DECLARE   x  FLOAT,
          i  FIXED ;
```

Zusammengefaßt können skalare Variablen folgendermaßen deklariert werden:

Skalare-Variablen-Deklaration ::=
{ **DECLARE** | **DCL** } Variablen-Angabe [, Variablen-Angabe] ··· ;

Variablen-Angabe ::=
Bezeichner-Angabe [Zuweisungsschutz] Typ-Attribut [Global-Attribut]
[Initialisierungsattribut]

Bezeichner-Angabe ::=
Bezeichner | (Bezeichner [, Bezeichner] ···)

Typ-Attribut ::=
einfacher-Typ | Typ-Referenz | Bezeichner§für-Typ

einfacher-Typ ::=
Typ-ganze-Zahl | Typ-Gleitpunktzahl |
Typ-Bitkette | Typ-Zeichenkette |
Typ-Uhrzeit | Typ-Dauer

Die allgemeine Form der Spezifikation skalarer Variablen lautet wie folgt:

Skalare-Variablen-Spezifikation ::=
{ **SPECIFY** | **SPC** } Variablen-Angabe-S [, Variablen-Angabe-S] ··· ;

Variablen-Angabe-S ::=
Bezeichner-Angabe [Zuweisungsschutz] Typ-Attribut
{ Global-Attribut | Identifikationsattribut }

Zuweisungsschutz und Initialisierungsattribut werden in 5.14 beschrieben; Identifikations- und Global-Attribut wurden bereits in 4.2 und 4.4 definiert.

Nachfolgend werden nun die verschiedenen Typ-Möglichkeiten vorgestellt; auf Variablen vom Typ Sema, Bolt, Interrupt oder Signal wird allerdings erst in 9.3, 9.4 und 10 eingegangen.

5.2 Ganze Zahlen (FIXED)

Ganze Zahlen können in Dezimaldarstellung oder Dualdarstellung geschrieben werden. Die Dezimaldarstellung einer ganzen Zahl besteht aus einer Folge von Ziffern. Die Dualdarstellung einer ganzen Zahl besteht aus einer Folge der Ziffern 0 und 1, die mit dem Zeichen B abgeschlossen wird.

Beispiele:

ganze Zahl	Dezimaldarstellung	Dualdarstellung
6	6	110B
123	123	1111011B

Zudem kann für eine Zahlenkonstante die Genauigkeit ihrer Darstellung definiert werden, indem anschließend an die Zahlenkonstante die Anzahl der Bitstellen in Klammern vermerkt wird, in denen sie rechnerintern ohne Vorzeichen dargestellt werden soll.

Beispiel: 123(31) Die ganze Zahl 123 wird in 31 Bitstellen dargestellt.

Ist keine Genauigkeit angegeben, wird die Genauigkeit aus der Konstanten abgeleitet.

Variablen für ganze Zahlen werden mit dem Typ-Attribut FIXED deklariert.

```
Typ-ganze-Zahl ::=
        FIXED  [ (Genauigkeit) ]

Genauigkeit ::=
        ganze-Zahl-ohne-Genauigkeit§größer-Null
```

Die Genauigkeit gibt die Anzahl der Bitstellen an, in denen der jeweilige Wert der Variablen (ohne Vorzeichen) dargestellt wird. Fehlt die Genauigkeitsangabe, so wird die in einer Längenvereinbarung (siehe 5.6) definierte Genauigkeit eingesetzt. Sinnvollerweise sollte als Genauigkeit 15 oder 31 gewählt werden.

Beispiel:

```
DCL     Zaehler   FIXED (31) ,
        (i, j, k)   FIXED ;
...
i := 2 ;
```

5.3 Gleitpunktzahlen (FLOAT)

Konstanten für Gleitpunktzahlen können dargestellt werden als eine Folge von

(i) einem Punkt, einer ganzen Zahl und eventuell einem Exponenten zur Basis 10, wobei ein Exponent aus einer Folge des Zeichens E, eventuell einem Plus- oder Minuszeichen und einer ganzen Zahl besteht.

Beispiel:
```
.123      (entspricht 0,123)
.123E2    (entspricht 12,3)
.123E-1   (entspricht 0,0123)
```

(ii) einer ganzen Zahl und der unter (i) angegebenen Folge.

Beispiel:
```
3.123E2   (entspricht 312,3)
```

(iii) einer ganzen Zahl, einem Punkt und eventuell einem Exponenten.

Beispiel:
```
3.        (entspricht 3,0)
```

(iv) einer ganzen Zahl und einem Exponenten

Beispiel:
```
3E-2      (entspricht 0,03)
```

Analog zu ganzen Zahlen können auch Konstanten für Gleitpunktzahlen mit Genauigkeit formuliert werden.

Variablen vom Typ Gleitpunktzahl (mit ganzen Zahlen oder Gleitpunktzahlen als Werten) werden mit dem Typ-Attribut FLOAT deklariert.

Typ-Gleitpunktzahl ::=
    FLOAT [(Genauigkeit)]

Es gelten die Aussagen von 5.2; sinnvolle Genauigkeiten sind hier 23 bzw. 53.

Beispiele:

```
DCL   (x, y, z)  FLOAT,
      Koeff  FLOAT (53) ;
...
x := 3.5 ;  y := 1 ;  Koeff := 3.14 E-10 ;
```

5.4 Bitketten (BIT)

Eine Bitkettenkonstante kann binär (B1), in Form von Tetraden (B2), Oktaden (B3) oder in hexadezimaler Form (B4) angegeben werden.

Die Form Bi (i = 1, ... , 4) besteht aus einem Apostroph, einer Folge

- der Ziffern 0 und 1 im Fall B1
- der Ziffern 0 bis 3 im Fall B2
- der Ziffern 0 bis 7 im Fall B3
- der Ziffern 0 bis 9 und Buchstaben A bis F im Fall B4

sowie einem Apostroph und der entsprechenden Angabe B1 oder B2 oder B3 oder B4.

Beispiel: '110010100111'B1 entspricht
'302213'B2 entspricht
'6247'B3 entspricht
'CA7'B4

Anstatt B1 kann kürzer B geschrieben werden. Die folgenden Tabellen zeigen die Zuordnung zwischen der Binärform und den anderen Formen:

B2	B1
0	00
1	01
2	10
3	11

B3	B1
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

B4	B1
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Variablen für Bitketten sind mit dem Typ-Attribut BIT zu vereinbaren.

Typ-Bitkette ::=
    BIT [(Länge)]

Länge ::=
    ganze-Zahl-ohne-Genauigkeit§größer-Null

Länge gibt die Anzahl Elemente der Bitkette an. Fehlt die Längenangabe, so muß sie entweder in einer Längenvereinbarung (siehe 5.6) definiert worden sein oder es wird als Länge 1 eingesetzt.

Beispiel:

```
DCL     X_Koord   BIT (2) ,
        Y_Koord   BIT (8) ;
...
X_Koord := '01'B ;
Y_Koord := 'A9'B4 ;
```

Die Ansprache von Teilen von Bitketten wird in 6.1, Ausdrücke, behandelt.

5.5 Zeichenketten (CHARACTER)

Eine Zeichenkettenkonstante besteht aus einem Apostroph, einer Folge von beliebigen Zeichen (außer Apostroph) und einem Apostroph.

Beispiel: 'Stoerung Nr: '

Soll jedoch die Zeichenkette einen Apostroph enthalten, so muß dieser durch zwei aufeinanderfolgende Apostrophe dargestellt werden.

Beispiel: 'Stoer' 'Nr: '

Steuerzeichen können mit Hilfe der Umschaltsymbole “ ‘\ “und “ \’ “ in Zeichenkettenkonstanten eingefügt werden. Die Steuerzeichen werden als Paare von Sedezimal-Ziffern angegeben.

Beispiele:
'Dieser String '\ 0D 0A \' enthält zwei Steuerzeichen'
''\1B\' Steuerzeichen am String-Anfang'
'Steuerzeichen am String-Ende'\00\''
''\00\''   /* String bestehend aus einem einzigen Steuerzeichen */

Beispiel:
'Durch die Umschaltung auf die Steuerzeichensequenz'\ 20
\'können sehr lange Zeichenkettenkonstanten'\ 20
\'(unabhängig vom verwendeten Editor) erstellt werden.'

Variablen für Zeichenketten deklariert man mit dem Typ-Attribut CHARACTER.

Typ-Zeichenkette ::=
{ **CHARACTER** | **CHAR** }   [(Länge)]

Länge gibt die Anzahl Zeichen an. Fehlt die Längenangabe, so muß sie entweder in einer Längenvereinbarung (siehe 5.6) definiert worden sein oder es wird die Länge 1 eingesetzt.

Beispiele:

DCL Artikelbez **CHAR** (6) ;
...
Artikelbez := 'BCD/27' ;

Die Ansprache von Teilen von Zeichenketten wird in 6.1, Ausdrücke, behandelt.

5.6 Die Längenvereinbarung

Mit der Längenvereinbarung werden die Genauigkeiten und Längen für solche Zahlen- und Ketten-Objekte definiert, deren Genauigkeiten und Längen nicht durch die Schreibweise (bei Konstanten) oder die Deklaration (bei Variablen) festgelegt ist.

```
Längenvereinbarung ::=
        LENGTH    {  { FIXED | FLOAT }  (Genauigkeit)
                  |  { BIT | CHARACTER | CHAR }  (Länge)   }  ;
```

Beispiel:

```
PROBLEM ;
        LENGTH   FIXED(15) ;
        LENGTH   FLOAT(53) ;
        DCL     A   FIXED ,             /* A hat den Typ FIXED(15) */
                X   FLOAT ,             /* X hat den Typ FLOAT(53) */
                Y   FLOAT(23) ;         /* Y hat den Typ FLOAT(23) */
        ...
```

Für eine Längenvereinbarung gelten die gleichen Gültigkeitsregeln wie für Variablendeklarationen (siehe 4.3, Blockstruktur).

5.7 Uhrzeiten (CLOCK)

Eine Uhrzeitkonstante besteht aus einer positiven ganzen Zahl für die Stundenangabe, einer ganzen Zahl zwischen 0 und 59 für die Minutenangabe und einer Gleitpunktzahl zwischen 0 und 59.999... für die Sekundenangabe - jeweils durch Doppelpunkt getrennt. Die Stundenangabe wird modulo 24 interpretiert.

Beispiel:

11:30:00	bedeutet 11.30 Uhr
15:45:3.5	bedeutet 15.45 Uhr und 3,5 Sekunden
25:00:00	bedeutet 1 Uhr

Variablen für Uhrzeiten werden mit dem Typ-Attribut CLOCK vereinbart.

```
Typ-Uhrzeit ::=
        CLOCK
```

Beispiel:

```
DCL   Zeit   CLOCK ;
...
Zeit := 12:30:00 ;
```

5.8 Zeitdauern (DURATION)

Eine Dauerkonstante setzt sich aus einer Stundenangabe, Minutenangabe und Sekundenangabe zusammen, wobei einzelne dieser Angaben fehlen können. Eine Stundenangabe besteht aus einer ganzen Zahl und der Zeichenfolge HRS, eine Minutenangabe aus einer ganzen Zahl und der Zeichenfolge MIN, eine Sekundenangabe aus einer Gleitpunktzahl und der Zeichenfolge SEC.

Beispiele:

5 **MIN** 30 **SEC**	bedeutet 5 Minuten und 30 Sekunden
.05 **SEC**	bedeutet 50 Millisekunden
5 **HRS** 10 **SEC**	bedeutet 5 Stunden und 10 Sekunden

Die ganzen Zahlen und Gleitpunktzahlen in Zeitkonstanten dürfen keine Genauigkeitsangaben enthalten.

Variablen für Zeitdauern müssen mit dem Typ-Attribut DURATION vereinbart werden.

```
Typ-Dauer ::=
        DURATION | DUR
```

Beispiel:

```
DCL  Verzoegerung  DUR ;
...
Verzoegerung := 0.1 SEC ;
```

5.9 Referenzen (REF)

Für die indirekte Adressierung stehen in PEARL sogenannte Referenz-Variablen (Zeigervariablen, Pointer) zur Verfügung. Referenz-Variablen haben als Werte die Namen von Variablen (sie zeigen auf Variablen). Analog zu den bisher eingeführten Variablen ist dabei der Wertebereich einer Referenz-Variablen auf einen Typ von Variablen beschränkt, der bei der Vereinbarung der Referenz-Variablen angegeben wird (siehe jedoch 5.13, Überlagerung von Datenstrukturen).

Auf den Wert einer Referenzvariablen, die referierte Variable, bezieht man sich mittels des monadischen Operators CONT (von "content"); dieser Vorgang wird "Dereferenzierung" genannt.

Beispiele:

```
DCL     (k, l)  FIXED,
        x   FLOAT,
        (rk1, rk2)   REF   FIXED,       /* Fixed-Referenz-Variable */
        rx   REF   FLOAT ;              /* Float-Referenz-Variable */
rk1 := k ;                /* rk1 zeigt auf k */
rk1 := l ;                /* rk1 zeigt auf l */
rk2 := rk1 ;              /* rk2 zeigt auf l, Zeigerzuweisung */
rx := x ;                 /* rx zeigt auf x */
rx := k ;                 /* falsch, Typ ungleich */
rx := rk1 ;               /* falsch, Typ ungleich */
l := 2 ;
k := CONT rk1 ;/* k erhält den Wert 2 */
rk2 := 3 ;                /* falsch, 3 ist keine Variable  */
CONT rk2 := 3 ;/* l erhält den Wert 3 */
CONT rk2 := k ;/* l erhält den Wert 2 */
```

Anstelle von k := CONT rk1 kann auch einfacher k := rk1 geschrieben werden; d.h. der Operator CONT darf auf der rechten Seite einer Zuweisung weggelassen werden ("implizite Dereferenzierung").

Allgemein gilt:

```
Typ-Referenz ::=
        REF  [ virt_Dimensionsliste ]  [ Zuweisungsschutz ]
        { einfacher-Typ | Bezeichner§für-Typ | Typ-Struktur |
          Typ-Dation | SEMA | BOLT | INTERRUPT | IRPT | SIGNAL |
          Typ-Prozedur | TASK | Typ-VOID | CHAR( )
        }

virt-Dimensionsliste ::=
        ( [ , ··· ] )
Dereferenzierung ::=
        CONT   Name§Referenz-Variable
```

Die Definition der Sprachform Typ-Referenz zeigt, welchen Typ eine Variable haben darf, auf die eine Referenz-Variable zeigen soll. Insbesondere dürfen Referenz-Variablen auch auf Felder und Strukturen zeigen (vgl. 5.10 und 5.11). Die Anzahl n der Kommata in der virtuellen Dimensionsliste gibt an, daß die referierte Variable ein (n + 1)-dimensionales Feld ist. Hingegen dürfen Referenz-Variablen nicht auf Referenz-Variablen zeigen; allerdings dürfen referierte Felder und Strukturen Referenz-Variablen als Elemente und Komponenten besitzen. Dies kann beispielsweise dazu benutzt werden, Strukturen miteinander zu verketten oder allgemein Listen aufzubauen. Punkt 5.12, Typvereinbarung, zeigt dazu ein Beispiel.

Typ-Struktur und Typ-Dation werden in 5.11 und 10.2 definiert, Typ-Prozedur und TASK in 8.3 und 9.1.1, Typ-VOID und REF CHAR( ) in 6.4.2 und 5.9.1.

Zeigt eine Referenz-Variable R auf ein Feld F mit den Elementen F (i, j, k, ...) oder auf eine Struktur S mit den Komponenten S.Ki, so werden (ausgehend von R) die Elemente von F oder die Komponenten von S ohne Benutzung von CONT mit R (i, j, k, ...) oder R.Ki angesprochen.

Eine Referenz-Variable wird außerdem implizit dereferenziert,

- wenn sie aktueller Parameter eines Prozeduraufrufs ist und der zugehörige formale Parameter keine Referenz-Variable ist.
- wenn sie als Operand eines dyadischen Operators benutzt wird, sofern dieser nicht gerade für Werte von Referenz-Variablen definiert ist wie IS (siehe unten).

Beispiel:

```
DCL   rk  REF  FIXED ,
      k   FIXED ;
      ...
      rk := k ;  k := 2 ;
      k := rk + 1 ;  /* aequivalent  k := k + 1 ; */
```

Um beispielsweise das Ende einer Kette zu kennzeichnen, steht eine Referenz-Variable NIL (Nullzeiger) zur Verfügung, die einen bestimmten, konstanten Wert besitzt.

Zum Vergleich der Werte von Referenz-Variablen können die dyadischen Operatoren IS oder ISNT benutzt werden; IS (ISNT) liefert das Ergebnis '1'B ('0'B), wenn die beiden angegebenen Referenz-Variablen denselben (verschiedene) Wert(e) besitzen, andernfalls das Ergebnis '0'B ('1'B).

Beispiel:

```
DCL   Nextauftrag   REF   Typ_Auftrag ;
...
IF   Nextauftrag   IS   NIL   THEN
        . . .
FIN ;
```

5.9.1 Variable Zeichenkettenzeiger (REF CHAR( ))

Variabel lange Zeichenketten können mit speziellen Zeiger-Objekten, den sogenannten variablen Zeichenkettenzeigern, leicht verarbeitet werden.

Beispiel: Deklarationen variabler Zeichenkettenzeiger

```
DCL str_ptr REF CHAR ( );                         /* Zeiger-Deklaration */
SPC print ENTRY ( REF CHAR ( ) ) GLOBAL;          /* Zeiger als Parameter */
```

Ein Zeiger vom Typ "REF CHAR ( )" enthält neben der Adresse einer Zeichenkette noch zwei Zähler, die die maximale Länge und die aktuell benutzte Länge der referenzierten Zeichenkettenvariable speichern. Bei der Zuweisung der Adresse einer Zeichenkettenvariable an einen variablen Zeichenkettenzeiger werden beide Zähler mit der Länge der referenzierten Zeichenkettenvariable initialisiert.

Beispiel: Initialisierung eines variablen Zeichenkettenzeigers

```
DCL s1        CHAR(20);
DCL s2        CHAR(100);
DCL str_ptr   REF CHAR ( );
str_ptr := s1;                /* Adr=s1,  Länge=20,   Max=20  */
str_ptr := s2;                /* Adr=s2,  Länge=100, Max=100 */
```

Damit unterscheidet sich der variable Zeiger zunächst nicht von einem Zeichenkettenzeiger mit einer konstanten Länge. Bei Zuweisungen an die Zeichenkettenvariable über diesen Zeiger wird jedoch die Länge der zugewiesenen Zeichenkette in dem Zähler für die aktuelle Länge gespeichert, und das sonst übliche Auffüllen der Zeichenkette bis zur maximalen Länge mit Leerzeichen entfällt.

Beispiel: Zuweisung an die referenzierte Variable

```
str_ptr := s1;                       /* Adr=s1,  Länge=20,   Max=20  */
CONT str_ptr := 'PEARL 90';          /* Adr=s1,  Länge=8,    Max=20  */

str_ptr := s2;                       /* Adr=s2,  Länge=100, Max=100 */
CONT str_ptr := 'Werum GmbH';        /* Adr=s2,  Länge=10,   Max=100 */
```

Bei den darauf folgenden Zugriffen auf die Zeichenkette über diesen Zeiger wird nur noch die aktuelle Länge benutzt.

Beispiel:

```
DCL s1        CHAR(20);
DCL s2        CHAR(100);          /* platzspendende Variable */
DCL str_ptr   REF CHAR ( );       /* variabler Zeiger */

str_ptr := s2;                                /* Adr=s2,  Länge=100, Max=100 */
CONT str_ptr := 'D21337 ' CAT 'Lueneburg';    /* Adr=s2,  Länge=16,   Max=100 */
```

Nach dieser Zuweisung enthält die Variable "s2" an den ersten 16 Stellen die Zeichen "D21337 Lueneburg", die restlichen Zeichen wurden durch die Zuweisung nicht verändert. Der Längenindex von "str_ptr" wurde auf die aktuelle Länge 16 eingestellt.

```
s1 := CONT str_ptr;
```

Bei dieser Zuweisung erzeugt "CONT str_ptr" den Zeichenkettenwert 'D21337 Lueneburg' (mit dem Typ "CHAR(16)"). Nach den Zuweisungsregeln von PEARL wird die Zeichenkette mit vier Leerzeichen erweitert, bevor dieser Wert der Variablen "s1" (mit dem Typ "CHAR(20)") zugewiesen wird.

Besonders nützlich erweisen sich variable Zeichenkettenzeiger als formale Parameter von Prozeduren. Der Compiler übergibt an der Aufrufstelle neben der Adresse einer Zeichenkettenvariablen auch deren aktuelle Länge. So kann z.B. eine Fehlerroutine unterschiedlich lange Fehlertexte ausgeben.

Beispiel: Zeichenkettenzeiger als Parameter

```
SPC print_fehler ENTRY ( REF CHAR ( ) ) GLOBAL;
DCL s1 CHAR(20)   INIT ( '...' );
DCL s2 CHAR(100) INIT ( '...' );

. . .
CALL print_fehler ( s1 );
CALL print_fehler ( s2 );
```

Es dürfen auch Zeichenkettenkonstanten übergeben werden, wenn der formale Parameter mit dem Typ "REF INV CHAR ( )" definiert wurde.

Beispiel:

```
SPC print_message ENTRY ( REF INV CHAR ( ) ) GLOBAL;

. . .
CALL print_message ( 'Text 1' );
CALL print_message ( 'laengerer Text 2' );
```

Ein variabler Zeichenkettenzeiger kann in allen Ausdrücken analog zu konstanten Zeichenkettenzeigern benutzt werden. Die zu der Typprüfung gehörende Längenprüfung erfolgt erst zur Laufzeit.

Ein variabler Zeichenkettenzeiger ist als Ergebniswert einer Funktion nicht erlaubt. Der Typ "CHAR ( )" darf nur in Verbindung mit "REF" benutzt werden, d.h. der Parametertyp "CHAR ( ) IDENT" ist nicht erlaubt.

5.10 Felder

Nach Möglichkeit werden beim Programmieren gleichartige Objekte unter einem Bezeichner zusammengefaßt und die einzelnen Objekte indiziert angesprochen.

Beispiel:

> Eine unterlagerte Steuerung steuert drei Geräte G(1), G(2) und G(3). Bei Ausgabe der Bitkette '0001'B an die unterlagerte Steuerung soll G(1) eingeschaltet werden, bei Ausgabe von '0010'B das Gerät G(2), bei Ausgabe von '0100'B das Gerät G(3). Hierzu bietet sich an, die drei Einschaltsignale unter einem Bezeichner, z.B. einschalten, zusammenzufassen und indiziert zu benutzen:

```
DCL    einschalten (1 : 3)   BIT (4) ,
       i   FIXED ;
einschalten (1) := '0001'B ;
einschalten (2) := '0010'B ;
einschalten (3) := '0100'B ;
...

/* Übernahme des Wertes von i (Index des einzuschaltenden Geräts) von einem
anderen Programmteil
Ausgabe von einschalten (i) an die unterlagerte Steuerung */
```

Allgemein können skalare Variablen gleichen Typs zu n-dimensionalen Feldern zusammengefaßt werden (n = 1, 2, 3, 4).

Dem Feld wird bei seiner Deklaration ein Bezeichner zugeordnet; die einzelnen Feldelemente (skalare Variablen) werden unter diesem Bezeichner und der Angabe ihrer Position innerhalb des Feldes (dem Index) angesprochen.

So wird z.B. mittels

```
DCL   Koeff (1:2, 0:3)   FIXED ;
```

ein 2-dimensionales Feld vereinbart, wobei die erste Dimension die untere Grenze 1 und die obere Grenze 2, d.h. die Länge 2 besitzt, während die zweite Dimension die untere Grenze 0 und die obere Grenze 3, d.h. die Länge 4 hat. Dies bedeutet, daß Koeff aus den 8 skalaren Fixed-Variablen

```
Koeff(1,0)     Koeff(1,1)     Koeff(1,2)     Koeff(1,3)
Koeff(2,0)     Koeff(2,1)     Koeff(2,2)     Koeff(2,3)
```

besteht.

Die allgemeine Form der Deklaration eines Feldes lautet:

Feld-Deklaration ::=
{ **DECLARE | DCL** } Feld-Angabe [, Feld-Angabe] ⋯ ;

Feld-Angabe ::=
Bezeichner-Angabe Dimensionsattribut [Zuweisungsschutz] Typ-Attribut
[Global-Attribut] [Initialisierungsattribut]

Dimensions-Attribut ::=
(Grenzangabe§erste-Dimension [, Grenzangabe§weitere-Dimension] ⋯)

Grenzangabe ::=
[Grenze§unten :] Grenze§oben

Grenze ::=
[-] ganze-Zahl-ohne-Genauigkeit | Bezeichner§benannte-Konstante |
Konstanter-Ausdruck

Demnach sind auch negative ganze Zahlen als Dimensionsgrenze zugelassen. Die obere Grenze einer Dimension muß jedoch immer größer oder gleich ihrer unteren Grenze sein.

Ist die untere Grenze nicht angegeben, so wird hierfür der Wert 1 angenommen. Unter einer benannten Konstante wird hier eine Variable vom Typ FIXED (15) verstanden, die mit Zuweisungsschutz (INV) und Initialwert (INIT) deklariert ist.

Beispiele:

```
DCL   einschalten (3)   BIT (4) ,
      Koeff (2, 0:3)   FIXED ,
      Meldung (20)   CHAR (12) ;
```

Das eindimensionale Feld Meldung enthält z.B. 20 Fehlertexte, damit im Fehlerfall i ein Programm die Meldung (i) an eine Konsole ausgeben kann (i = 1, ... , 20).

Felder können auch gemischt mit skalaren Variablen in einer Deklaration vereinbart werden, z.B.:

DCL Anz_Meld **INV** **FIXED** (15) **INIT** (20) ;
...

DCL Meldung (Anz_Meld) **CHAR** (12) ,
(i, j, k) **FIXED** ,
(einschalten, ausschalten) (3) **BIT** (4) ;

Die allgemeine Form der Spezifikation eines Feldes lautet:

Feld-Spezifikation ::=
{ **SPECIFY** | **SPC** } Feld-Angabe-S [, Feld-Angabe-S] ··· ;

Feld-Angabe-S ::=
Bezeichner-Angabe virtuelles-Dimensionsattribut [Zuweisungsschutz]
Typ-Attribut { Global-Attribut | Identifikationsattribut }

virtuelles-Dimensionsattribut ::=
([, ···])

Die Anzahl n der Kommata im virtuellen Dimensionsattribut gibt an, daß das spezifizierte Feld n + 1 Dimensionen besitzt.

In Punkt 2 des Anhangs ist tabellarisch beschrieben, welche Größen zu Feldern zusammengefaßt werden dürfen.

5.11 Strukturen

Felder erlauben die Zusammenfassung skalarer Variablen des *gleichen* Typs unter einem Bezeichner; die einzelnen Variablen werden mit diesem Bezeichner und ihrem Index angesprochen.

Bei vielen Automationsaufgaben, insbesondere bei solchen mit dispositivem Charakter, müssen jedoch Datenstrukturen beschrieben werden, deren Komponenten *unterschiedliche* Typen besitzen.

Beispiel:

Die Beiträge für die Fernseh-Nachrichtensendungen eines Tages sind auf Magnetaufzeichnungsgeräten (MAZ) gespeichert; die Beitragsfolge einer bestimmten Nachrichtensendung dieses Tages wird von Redakteuren mit Hilfe eines Rechners zusammengestellt, der die entsprechenden MAZ steuert. Dazu ist für jeden Beitrag ein Datensatz nötig, der z.B. folgende Struktur besitzt:

- Identitätskennzeichen des Beitrages
- Archivnummer
- Hinweise, ob der Beitrag bereits in der ersten, zweiten oder dritten Nachrichtensendung dieses Tages gesendet wurde
- Anfangsposition des Beitrages auf dem Band
- Endeposition des Beitrages auf dem Band
- Kennzeichen, ob Originalton vorhanden ist
- Länge des gegebenenfalls zu speichernden Texts
- Ein gegebenenfalls zu speichernder Text

Solche Datenstrukturen lassen sich problemorientiert als Struktur beschreiben; die obige Struktur "Beitrag" wird beispielsweise wie folgt vereinbart:

```
DCL  Beitrag  STRUCT
   [ (Identnr, Archiv)  FIXED,
      schon_gesendet (3)  BIT (1),
     (Anfang, Ende)  FIXED,
     Originalton  BIT (1),
     Textlaenge  FIXED,
     Text  CHAR (200)  ] ;
```

(Eine Struktur wird in eckige Klammern eingeschlossen; diese werden hier im Text fett gedruckt, um sie von den metasprachlichen Zeichen “[“ und “]“ zu unterscheiden. In der Programmniederschrift werden nur die eckigen Klammern oder ihre Ersatzsymbole “(/“ und “/)“ benutzt.)

Anders als Feldelemente werden die Komponenten einer Struktur nicht über den gemeinsamen Bezeichner und einen Index, sondern über den gemeinsamen Bezeichner und ihren in der Vereinbarung angegebenen Bezeichner angesprochen, wobei die beiden Bezeichner durch einen Punkt getrennt werden.

Beispiel:

Durch die Anweisung

```
Beitrag.Anfang := 1027;
```

erhält die Komponente Anfang der Struktur Beitrag den Wert 1027 zugewiesen.

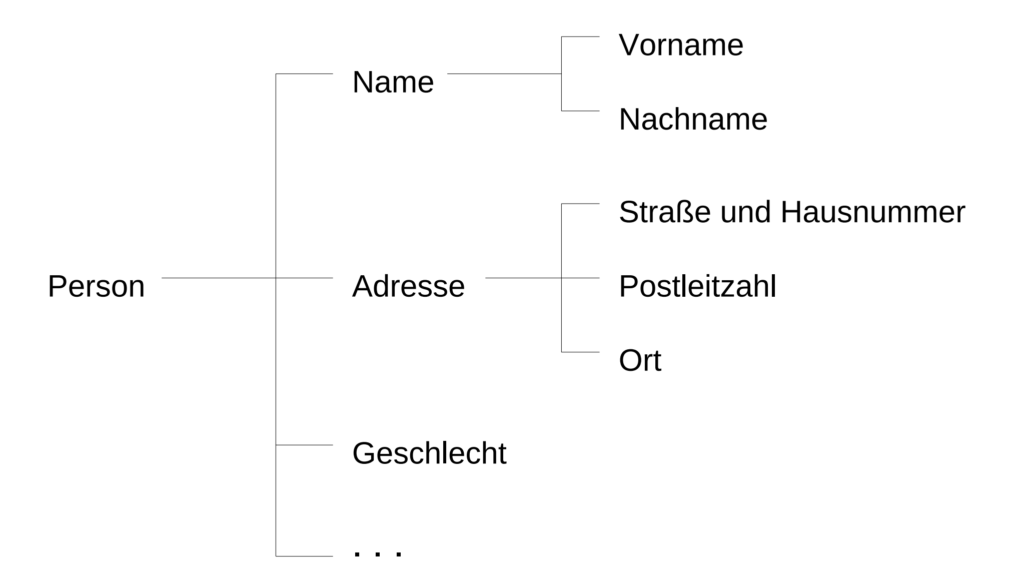
Wie die Vereinbarung von Beitrag zeigt, dürfen die Komponenten von Strukturen Felder sein; die Feldelemente werden dann wie gewohnt mit ihrem indizierten Bezeichner angesprochen, dem der Strukturbezeichner vorangestellt ist, etwa in der Form

IF Beitrag.schon_gesendet (i) **THEN** ... **FIN** ;

Eine Strukturkomponente kann jedoch auch selbst wieder eine Struktur sein, wodurch nicht nur lineare, sondern auch hierarchische Datenstrukturen nachgebildet werden können.

Beispiel:

Eine Personaldatei enthalte die Beschreibungen von Mitarbeitern; diese Beschreibungen haben jeweils folgende Struktur:



Die Komponenten Name und Adresse sind selbst Strukturen, nämlich Unterstrukturen der Hauptstruktur Person; diese kann so vereinbart werden:

```
DCL   Person   STRUCT
             [  Name       STRUCT
                        [  Vorname    CHAR (10),
                           Nachname   CHAR (15)
                        ] ,
                Adresse    STRUCT
                        [  Str    CHAR (15);
                           PLZ    FIXED,
                           Ort    CHAR (15)
                        ] ,
                Geschlecht  CHAR (1),
                . . .
             ] ;

. . .
```

Unterstrukturen können also über ihren Bezeichner und die vorangestellten Bezeichner der jeweils übergeordneten Strukturen angesprochen werden.

Der Typ einer Struktur wird durch die Anordnung ihrer Komponenten und deren Typ bestimmt. Strukturen gleichen Typs können zu Feldern zusammengefaßt werden:

```
DCL   Beitrag(20)   STRUCT ... ;  /* siehe oben */
```

Allgemein wird deshalb eine beliebige Komponente einer Struktur unter ihrem (gegebenenfalls indizierten) Bezeichner und den (gegebenenfalls indizierten) Bezeichnern aller übergeordneten Strukturen angesprochen, wobei diese Namen jeweils durch einen Punkt getrennt werden.

Beispiel:

```
Person.Name.Vorname := 'OTTO' ;
IF   Beitrag(i).schon_gesendet (j)   THEN ... FIN ;
```

Die allgemeine Form der Struktur-Deklaration lautet:

Struktur-Deklaration ::=
{ **DECLARE** | **DCL** } Struktur-Angabe [, Struktur-Angabe] ··· ;

Struktur-Angabe ::=
Bezeichner-Angabe§Hauptstruktur [Dimensionsattribut] [Zuweisungsschutz]
Typ-Struktur [Global-Attribut] [Initialisierungsattribut]

Typ-Struktur ::=
STRUCT **[** Strukturkomponente [, Strukturkomponente] ··· **]**

Strukturkomponente ::=
Bezeichner-Angabe Typ-Attribut-in-Struktur-Vereinbarung

Typ-Attribut-in-Struktur-Vereinbarung ::=
[Dimensionsattribut]
{ einfacher-Typ | Typ-Referenz | Bezeichner§für-Typ | Typ-Struktur }

Die allgemeine Form der Struktur-Spezifikation lautet:

Struktur-Spezifikation ::=
{ **SPECIFY** | **SPC** } Struktur-Angabe-S [, Struktur-Angabe-S] ··· ;

Struktur-Angabe-S ::=
Bezeichner-Angabe§Hauptstruktur [virtuelles-Dimensionsattribut]
[Zuweisungsschutz] Typ-Struktur { Global-Attribut | Identifikationsattribut }

5.12 Typvereinbarung (TYPE)

Eine bestimmte Struktur kann beispielsweise als Parameter oder als Unterstruktur in anderen Strukturen oder als Typ der Übertragungsdaten einer Datenstation auftreten. Dabei muß jedesmal der Typ dieser Struktur angegeben werden, was bei komplexen Strukturen umfangreichen Schreibaufwand erfordern kann. Deshalb und zur Erhöhung der Lesbarkeit des Programms kann der Typ einer Struktur unter einem frei wählbaren Bezeichner als neuer Datentyp vereinbart und unter diesem Bezeichner z.B. dazu benutzt werden, Variablen dieses Datentyps zu vereinbaren.

Beispiele:

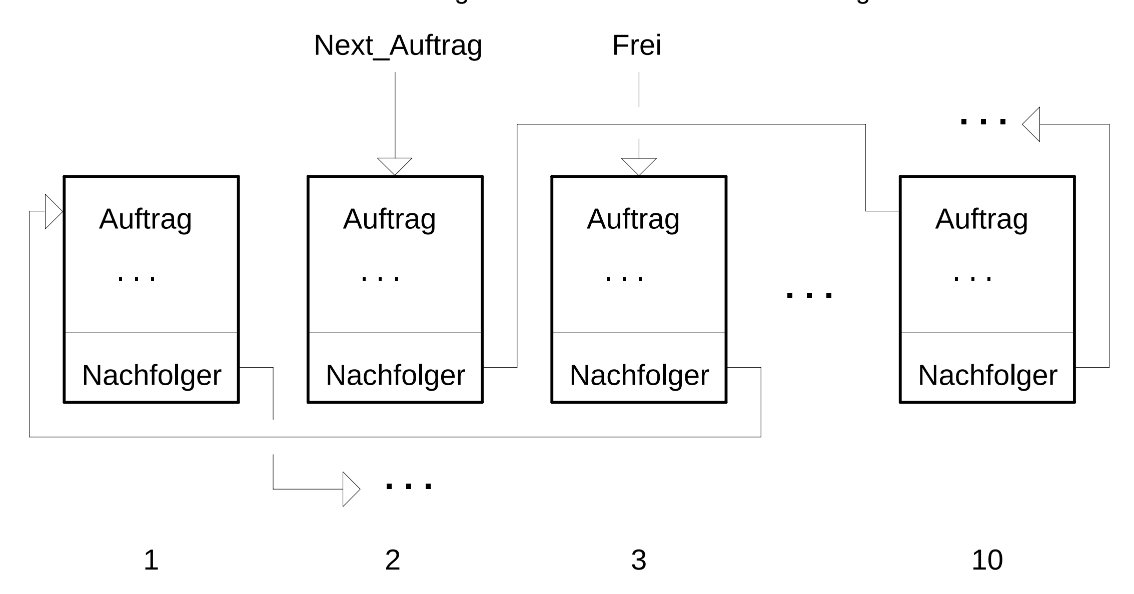
(1)

```
PROBLEM ;
        ...
        TYPE   Nachricht   STRUCT
             [ (Identnr, Archiv)   FIXED,
               schon_gesendet (3)   BIT (1),
               (Anfang, Ende)   FIXED,
               Originalton   BIT (1),
               Textlaenge   FIXED,
               Text   CHAR (200)   ] ;

        DCL   Inhalt_MAZ   DATION   INOUT   Nachricht ... ;

        Koord:  TASK ;
                DCL   Beitrag Nachricht ;
                ...
                READ   Beitrag   FROM   Inhalt_MAZ ;
                ...
                END ; ! Koord
        ...
```

(2) Eine Task steuert bestimmte Geräte; die Steuerungsaufträge erhält sie von einer anderen Task. Weil zeitweise mehr als ein Auftrag vorliegen kann und die Aufträge unterschiedliche Dringlichkeit haben können, werden sie in Form einer Warteschlange gepuffert. Die Struktur der Aufträge sei vom (neu vereinbarten) Typ Auftragstyp; dieser enthalte auch das Dringlichkeitskriterium, nach dem neue Aufträge eingeordnet werden. Die Referenz-Variablen Next_Auftrag bzw. Frei zeigen auf den nächsten zu bearbeitenden Auftrag bzw. den nächsten freien Platz in der Warteschlange Auftragskette; ausgehend von Frei seien die freien Plätze ebenfalls verkettet. Die Warteschlange sei höchstens 10 Elemente lang.



Die Auftragskette kann folgendermaßen vereinbart werden:

```
. . .
TYPE    Kettenelement   STRUCT
                   [    Auftrag     Auftragstyp,
                        . . . ,
                        Nachf       REF   Kettenelement
                   ] ;
DCL     Auftragskette  ( 10 )   Kettenelement,
        (Next_Auftrag, Frei)   REF   Kettenelement ;
. . .
```

Nach erfolgter Initialisierung und dem Einordnen verschiedener Aufträge kann ein fertig bearbeiteter Auftrag wie folgt aus der Warteschlange entfernt werden (Hilf sei dabei eine Variable vom Typ REF Kettenelement).

```
...
Hilf := Next_Auftrag ;
Next_Auftrag  := Next_Auftrag.Nachf ;
Hilf.Nachf := Frei ;
Frei := Hilf ;
...
```

Diese Anweisungen sind auch weitere Beispiele für implizites Deferenzieren. Die Auftragskette kann so initialisiert werden:

```
FOR  i  TO  9
        REPEAT
                Auftragskette (i).Nachf := Auftragskette (i + 1) ;

        END ;
Auftragskette (10).Nachf := NIL ;
```

Allgemein wird ein neuer Datentyp wie folgt vereinbart:

```
Typ-Vereinbarung ::=
        TYPE  Bezeichner§für-Typ  { einfacher-Typ | Typ-Struktur }
```

Die Vereinbarung eines neuen Datentyps muß vor seiner Benutzung erfolgen. Im übrigen können auf Modulebene vereinbarte Datentypen in allen Tasks und Prozeduren des Moduls benutzt werden; die Benutzbarkeit lokal vereinbarter Datentypen richtet sich nach den Regeln der Blockstruktur.

Im Gegensatz zu globalen Objekten können neu vereinbarte Datentypen nur in dem Modul benutzt werden, in dem sie vereinbart sind. Typvereinbarungen für globale Objekte müssen also in allen benutzenden Moduln wiederholt werden.

Beispiel:

```
MODULE  (Modul A) ;
PROBLEM ;

    TYPE   T   STRUCT  [ . . . ] ;

    . . .

    DCL  x   T  GLOBAL ;

     . . .

MODEND ;
```

```
MODULE  (Modul B) ;
PROBLEM ;

    TYPE   T   STRUCT  [ . . . ] ;

    . . .

    SPC  x   T  GLOBAL ;

     . . .

MODEND ;
```

5.13 Das Initialisierungsattribut (INITIAL)

Das Initialisierungsattribut erlaubt, Variablen bei ihrer Deklaration Anfangswerte zuzuweisen. Für Variablen mit Zuweisungsschutz (siehe 5.15) ist dies die einzige Möglichkeit der Wertzuweisung. Das Initialisierungsattribut wird bei der jeweiligen Deklaration als letztes Attribut angegeben.

```
Initialisierungsattribut ::=
        { INITIAL | INIT } ( Init-Element [ , Init-Element ] ··· )

Init-Element ::=
          Konstante
        | Bezeichner§benannte-Konstante
        | konstanter-Ausdruck
        ...

Konstante ::=
          ganze-Zahl
        | Gleitpunktzahl
        | Bitkettenkonstante
        | Zeichenkettenkonstante
        | Uhrzeitenkonstante
        | Dauerkonstante
        | NIL
```

Der Typ des angegebenen Init-Elementes muß mit dem Typ der deklarierten Variablen gemäß den Regeln für die Zuweisung (vgl. 6.3) verträglich sein.

Beispiel:

```
...
DCL  Anzahl_Geraete  FIXED  INIT(12),
        (UGR, OGR)  FIXED  INIT(2,15) ;
...
FOR  i  FROM  1  TO  Anzahl_Geraete
...
```

Hier haben die Variablen Anzahl_Geraete, UGR und OGR die Werte 12, 2 und 15, d.h. die Elemente einer angegebenen Konstanten-Liste werden in der Reihenfolge der Niederschrift der angegebenen Bezeichner-Liste zugeordnet.

Felder und Strukturen können ebenfalls an der Deklarationsstelle initialisiert werden. Für alle Strukturkomponenten muß jeweils ein Initialwert angegeben werden. Bei Feldern darf die Initialisierungsliste kürzer als die Anzahl der Feldelemente sein. In diesem Fall wird der letzte Initialwert für die restlichen Elemente wiederholt benutzt.

Beispiel:

```
...
DCL  Adresse  STRUCT [    Postleitzahl    FIXED,
                          Ort             CHAR(20),
                          Strasse         CHAR(20),
                          HausNr          FIXED        ]
              INIT (  21337 ,  'Lueneburg' ,  'Erbstorfer Landstr.' , 14 ) ;
DCL  Farben          ( 3 )  CHAR(7)       INIT (  'schwarz' ,  'weiss  ' ,  'rot    ' ) ;
DCL  alles_Sechsen   ( 6 )  FIXED         INIT (  6  ) ;
...
```

Die Initialisierungsliste kann außer Konstanten auch Bezeichner von benannten Konstanten, sowie vom Compiler berechenbare Ausdrücke enthalten (vgl. 5.15.1 und 5.15.2).

5.14 Zuweisungsschutz (INV)

Variablen können mit einem Zuweisungsschutz, dem Attribut INV (von "invariant"), deklariert werden, um - abgesehen von der Initialisierung - Zuweisungen an diese Variablen zu verbieten. Das Attribut INV wird unmittelbar vor dem Typ-Attribut angegeben.

Beispiel:

```
...
DCL    Pi  INV  FLOAT  INIT (3.141) ;

Pi := 3 ;  /* Ergibt Fehlermeldung des Compilers */
```

Ein einmal für ein Objekt in seiner Deklaration vereinbarter Zuweisungsschutz kann nicht aufgehoben werden; er muß deshalb in Spezifikationen dieses Objekts berücksichtigt werden, aber auch bei Übertragungen als aktueller Prozedurparameter oder bei der Benutzung als Wert von Referenz-Variablen. Andererseits darf auf eine dieser Arten ein bei der Deklaration noch nicht vereinbarter Zuweisungsschutz entstehen.

Beispiel:

```
...
P :    PROC ( A  (,)  INV  FIXED  IDENT , x  FLOAT  IDENT ) ;
              /* Prozedurkörper */
       END ; ! P

DCL    Tab (10,20)  FIXED,
       Pi  INV  FLOAT  INITIAL (3.141),
       R1  REF  FLOAT,
       R2  REF  INV  FIXED ;
 ...

SPC  pi  FLOAT  IDENT  (Pi) ;        /* Falsch */
...

CALL P (Tab, Pi) ;                   /* Falsch */
R1 := Pi ;                           /* Falsch */
R2 := Tab (5, 7) ;
```

Die Spezifikation von pi ist falsch, weil Zuweisungen an pi erlaubt werden und somit der Zuweisungsschutz von Pi unterlaufen wird.

Der Aufruf von P ist falsch, weil der zu Pi gehörige formale Parameter x ohne Zuweisungs-schutz INV vereinbart ist und somit im Körper von P über eine Zuweisung an x eine (verbotene) Wertänderung von Pi erfolgen könnte.

Dagegen ist im selben Aufruf die Parameterübertragung von Tab an A korrekt, da lediglich ein bisher noch nicht vereinbarter Zuweisungsschutz für Tab entstanden ist.

Die Zuweisung R1 := Pi ist unzulässig, da andernfalls der Zuweisungsschutz für Pi durch (erlaubte) Zuweisungen an CONT R1 aufgehoben werden könnte. Die Zuweisung
R2 := Tab(5,7) ist dagegen zulässig; eine Wertänderung des Tabellenelementes über CONT R2 ist nicht möglich.

5.14.1 Benannte Konstanten (INV-Konstanten)

Mit dem Attribut INV deklarierte Objekte vom Typ FIXED, BIT, CHAR(1), CLOCK und DURATION behandelt der Compiler als (benannte) Konstanten. Die Namen dieser Objekte dürfen nach ihrer Deklaration (mit Initialisierung) bei der Deklaration und Initialisierung weiterer Objekte benutzt werden. Dabei dürfen sie sowohl als Dimensionsgrenzenangabe bei Felddeklarationen, als auch in Initialisierungslisten eingesetzt werden.

Beispiel:

Ein Überwachungsprogramm soll für verschiedene Projekte mit unterschiedlichen Anzahlen von unterlagerten Steuerungen angepaßt werden. Die Anzahl der Nachrichtenpuffer und zugehörigen Synchronisiervariablen (vergl. 9.3) lassen sich durch Ändern der Konstante AnzahlSteuerungen leicht anpassen:

```
...
DCL   AnzahlSteuerungen   INV   FIXED   INIT (11) ;

DCL   NachrichtenPuffer   ( AnzahlSteuerungen )   CHAR(100);
DCL   Pufferzugriff    ( AnzahlSteuerungen )   BOLT;
...
```

Für ein weiteres Projekt mit einer anderen Anzahl von Steuerungen muß nur die benannte Konstante AnzahlSteuerungen den aktuellen Gegebenheiten angepaßt werden. Die davon abhängigen Speicherbereiche werden durch den Compiler angepaßt.

Da der Compiler für die benannten Konstanten keine Laufzeit-Objekte erzeugt, dürfen sie nicht einer Referenz-Variablen zugewiesen werden. Ebenso können sie nicht als aktueller Parameter einer Prozedur benutzt werden, wenn der formale Parameter mit dem IDENTICAL-Attribut oder als Referenzvariable spezifiziert wurde (vergl. 8.2).

5.14.2 Konstante Ausdrücke (vom Typ FIXED)

An allen Stellen, an denen der Compiler Konstanten erwartet, dürfen auch konstante Ausdrücke stehen. Diese Ausdrücke dürfen als Operanden nur Konstanten (auch benannte Konstanten) enthalten. Sie werden durch den Compiler ausgewertet und das Ergebnis an den entsprechenden Stellen eingesetzt.

```
konstanter-Ausdruck ::=
             { + | - } Gleitpunktzahl
          |  { + | - } Dauerkonstante
          |  konstanter-FIXED-Ausdruck

konstanter-FIXED-Ausdruck ::=
          Term  [ { + | - }  Term  ] ...

Term ::=
          Faktor  [ { * | // | REM }  Faktor  ] ...

Faktor ::=
          [ + | - ]
          {    ganze-Zahl
             |  (  konstanter-FIXED-Ausdruck  )
             |  TOFIXED  {  Zeichenkettenkonstante§der-Länge-1  |  Bitkettenkonstante  }
             |  Bezeichner§benannte-Konstante
          }
          [  FIT  konstanter-FIXED-Ausdruck  ]
```

Konstante Ausdrücke vom Typ FIXED können z.B. an folgenden Stellen verwendet werden:

- Angabe von Dimensionsgrenzen bei Felddeklarationen,
- Genauigkeitsangaben (bei FIXED- und FLOAT-Objekten),
- Längenangaben (bei CHAR- und BIT-Objekten),
- in Initialisierungslisten (auch bei PRESET),
- Prioritätsangaben bei Task-Deklarationen/-Anweisungen und
- in den Konstantenlisten der CASE-Anweisung
- auf der rechten Seite einer Zuweisung.

Oft sind mehrere Objekte eines Programmes (Variablen, Konstanten) voneinander abhängig. Läßt sich diese Abhängigkeit durch eine einfache Formel beschreiben, kann die Konfigurierung eines Programmes evtl. durch Anpassen einiger weniger Konstanten erledigt werden.

Beispiel:

Es soll eine Warnung ausgegeben werden, wenn ein Puffersystem bis auf die letzten 10% ausgefüllt ist:

```
...
DCL   MaxPuffer           INV    FIXED   INIT ( 1000 ) ;
DCL   WarnGrenze          INV    FIXED   INIT ( MaxPuffer  -  MaxPuffer // 10 ) ;
DCL   Puffer    ( MaxPuffer )    FIXED ;
DCL   aktuellerPufferIndex       FIXED   INIT ( 1 ) ;
...
aktuellerPufferIndex := aktuellerPufferIndex + 1 ;
IF aktuellerPufferIndex   >   WarnGrenze
THEN   /* Warnung ausgeben */
FIN ;
...
```

6. Ausdrücke und Zuweisungen

6.1 Ausdrücke

Ausdrücke werden bei verschiedenen Sprachformen benutzt, etwa

- als Index bei der Ansprache von Feldelementen

  Bezeichner§Feld (Ausdruck, Ausdruck)

  z.B. Tab (K, 2*i)

- als Referenzwert bei der Return-Anweisung in Funktionsprozeduren (siehe 8.1)

  RETURN (Ausdruck) ;

  z.B. **RETURN** (Nr) ;

- als Parameter beim Aufruf von Prozeduren (siehe 8.2)

  Liste-akt-Parameter ::= (Ausdruck [, Ausdruck] ···)

  z.B. **CALL** P (A, Tab (K , 2*i)) ;

- als Startbedingung für Tasks (siehe 9.2.1)

  AT Ausdruck§Uhrzeit

  z.B. **AT** 12:00:00 **ACTIVATE** T ;

Diese Beispiele zeigen, daß ein Ausdruck zumindest

- eine Konstante
- ein Bezeichner
- ein indizierter Bezeichner oder
- ein arithmetischer Ausdruck (z.B. 2*i)

sein kann.

Bezeichner, indizierte Bezeichner und Namen von Strukturkomponenten werden allgemein unter dem Begriff "Name" zusammengefaßt:

Name ::= Bezeichner [(Index [, Index] ···)] [. Name] ···

Index ::= Ausdruck§mit-ganzer-Zahl-als-Wert

Beispiele:

```
A,   A(3),   A(i, j, 2*k),   A.B,   A.B.C,   A(3).B.C(i, j)
```

Die Bezeichner der Komponenten einer Struktur können unabhängig von den Bezeichnern außerhalb der Struktur gewählt werden:

```
DCL     Person   STRUCT ...,  /* siehe oben */
        Ort   CHAR (15) ;
        ...
Person.Adresse.Ort := Ort;
```

Die in Ausdrücken angegebenen Namen müssen im allgemeinen Namen von skalaren Variablen sein; in den Ausdruckslisten der Ausgabe-Anweisungen (vgl. 10.4, 10.5, 10.6, 10.7) dürfen jedoch auch Bezeichner von Feldern angegeben werden.

Allgemein hat ein Ausdruck die Form

```
Ausdruck ::=
        [ monadischer-Operator ] Operand [ dyadischer-Operator   Ausdruck ] ...
```

Monadische Operatoren haben nur einen Operanden, dyadische Operatoren haben zwei Operanden.

```
monadischer-Operator ::=
          + | - | NOT | ABS | SIGN | LWB | UPB | SIZEOF | CONT
        | TOFIXED | TOFLOAT | TOBIT | TOCHAR | ENTIER | ROUND
        | Bezeichner§Benutzer-definierter-monadischer-Operator

dyadischer-Operator ::=
          + | - | * | / | // | REM | ** | < | LT | > | GT | <= | LE | >= | GE | == | EQ | /= | NE
        | AND | OR | EXOR | >< | CAT | <> | CSHIFT | SHIFT | LWB | UPB | IS | ISNT | FIT
        | Bezeichner§Benutzer-definierter-dyadischer-Operator

Operand ::=
        Konstante | Name | Funktionsaufruf | bedingter-Ausdruck | Dereferenzierung |
        Kettenausschnitt | (Ausdruck)
```

Beispiele:

- -A + B * C - D/E**2
- (A - B) / (A + B)
- F(Tab(K, 2*i)) / (F(i) - 3)
- A < B OR A < C
- Prozessabbild_neu   AND NOT   Prozessabbild_alt
  (das Ergebnis hat an *der* Bitstelle eine 1, an der Prozessabbild_alt eine 0 und Prozessabbild_neu eine 1 besitzt)
- Xkoord >< Ykoord >< Zkoord
  (die drei Bitketten werden zu einer Bitkette verkettet)

Der bedingte Ausdruck kann z.B. in Zuweisungen oder Funktionsprozeduren (siehe 8) nützlich sein; er hat folgende Form:

```
bedingter-Ausdruck ::=
    IF   Ausdruck§mit-einem-Wert-vom-Typ-BIT (1)
        THEN   Ausdruck   ELSE   Ausdruck   FIN
```

Ergibt die Berechnung des Ausdrucks hinter IF den Wert '1'B (wahr), so wird der Ausdruck hinter THEN anstelle des bedingten Ausdrucks eingesetzt; ist der Wert gleich '0'B (falsch), so wird der Ausdruck hinter ELSE eingesetzt.

Beispiele:

(1) Die Funktionsprozedur max soll die größere von zwei Gleitpunktzahlen feststellen und zurückgeben.

```
max:    PROC ( ( X, Y) FLOAT)   RETURNS   (FLOAT) ;
        RETURN (IF   X < Y   THEN   X   ELSE   Y   FIN) ;
        END ;
...
A := max (B, C)/2.0 ;
```

(2) Gleichwertig mit dieser Zuweisung ist die folgende Zuweisung:

```
A := (IF   B > C   THEN   B   ELSE   C   FIN) / 2.0 ;
```

Die Bedeutung der Dereferenzierung wurde unter 5.9, Referenzen, behandelt. Die allgemeine Form lautet:

Dereferenzierung ::=
    CONT { Name§Referenz **|** Funktionsaufruf }

Das i-te Bit einer Bitkette kann mittels des standardmäßigen Namens BIT(i) angesprochen werden, der - durch einen Punkt getrennt - hinter dem Namen der Bitkette angegeben wird. Eine Bitkette B der Länge lg wird also als Struktur B aufgefaßt, die als einzige Komponente einen eindimensionalen Bereich BIT der Länge lg besitzt, dessen Elemente vom Typ BIT(1) sind.

Analog hierzu kann das i-te Zeichen einer Zeichenkette Z mit Z.CHAR(i) oder Z.CHARACTER(i) angesprochen werden.

Dabei werden die Bits oder Zeichen einer Bit- oder Zeichenkette von links nach rechts numeriert.

Beispiel:

```
...
DCL     Byte   BIT(8),
        Bt   BIT(1),
        i   FIXED ;
...
Byte := '11101111'B ;
Bt := Byte.BIT(4) ;        /* Bt erhaelt den Wert '0'B */
Byte.BIT(2) := '0'B ;      /* Byte hat den Wert '10101111'B */
i := 8 ;
Byte.BIT(i) := Bt ;        /* Byte hat den Wert '10101110'B */
```

Darüber hinaus können mehrere Bits oder Zeichen umfassende Ausschnitte von Bit- oder Zeichenketten in analoger Art und Weise angesprochen werden; die allgemeine Form der Ansprache ist:

Kettenausschnitt ::=
    Name§Kette **.** { **BIT | CHAR | CHARACTER** } (Ausdruck§Anfang [: Ausdruck§Ende])

Die Ausdrücke Anfang und Ende müssen ganzzahlige Werte haben. Ende muß größer oder gleich Anfang sein, und beide Werte müssen innerhalb der deklarierten Kettenlänge liegen.

Das Ergebnis eines solchen Kettenausschnitt-Ausdrucks hat den Typ BIT(lg), bzw. CHAR(lg), wobei "lg := Ende - Anfang + 1" gilt. Bei Bitketten muß der Compiler die Länge "lg" errechnen können, damit weitere Typprüfungen bei der Verwendung des Kettenausschnitt-Ausdrucks möglich sind.

Folgende Fälle werden vom Compiler unterschieden:

(1) Name **.** { BIT | CHAR } (Ausdruck)

(2) Name **.** { BIT | CHAR } (konstanter-FIXED-Ausdruck **:** konstanter-FIXED-Ausdruck)

(3) Name **.** { BIT | CHAR } (Ausdruck1 **:** Ausdruck2 + FIXED-Konstante)

(4) Name **.** CHAR (Ausdruck§Anfang **:** Ausdruck§Ende)

Im Fall (1) ist die Länge des Kettenausschnitts gleich 1, “Ausdruck“ bezeichnet die Bit-Nr. bzw. die Character-Position.

Im Fall (2) errechnet der Compiler die Werte der beiden konstanten Ausdrücke und ermittelt daraus die Länge "lg := Konstante§Ende - Konstante§Anfang + 1".

Im Fall (3) müssen Ausdruck1 und Ausdruck2 gleich sein. Dann ergibt sich die Länge aus der additiven Konstanten: "lg := FIXED-Konstante + 1".

Im Fall (4) kann der Compiler die Länge des Kettenausschnitts nicht berechnen! Dies ist für Bitketten nicht erlaubt. Der Fall (4) liefert eine variable Zeichenkette, d.h. die Länge ist erst zur Laufzeit berechenbar. Variable Zeichenketten können außer in Verbindung mit dem CAT-Operator überall benutzt werden.

Beispiel:

Ein Regalförderzeug sei über 16 aufeinanderfolgende Stellen an einen Digitaleingang angeschlossen. Die Bedeutung der Anschlußstellen sei wie folgt:

Anschluß 1 - 8	:	X-Koordinate
Anschluß 9 - 12	:	Y-Koordinate
Anschluß 13	:	Z-Koordinate
Anschluß 14 - 16	:	weitere Parameter

Nach erfolgter Positionierung soll die Ist-Position abgefragt und überprüft werden.

```
MODULE ;
SYSTEM ;
    RFZ : DIGE(1)*0* 1, 16 ;
    ...
PROBLEM ;
    SPC    RFZ   DATION   IN   BIT(16) ;
    DCL    Rfz   DATION   IN   BIT(16)   CREATED ( RFZ ) ;
    ...
    Steuerung : TASK ;
        DCL       Rfz_Zustand   BIT(16) ,
                  Xkoord   BIT(8) ,   Ykoord   BIT(4) ,   Zkoord   BIT(1) ;
        ...
                  /* Positionierung */
        TAKE   Rfz_Zustand   FROM   Rfz ;
        Xkoord := Rfz_Zustand.BIT (1:8) ;
        Ykoord := Rfz_Zustand.BIT (9:12) ;
        Zkoord := Rfz_Zustand.BIT (13) ;
                  /* Überprüfung der Ist-Position */
        ...
    END ; ! Steuerung
    ...
```

Dieses Programmstück kann flexibler programmiert werden, indem zusätzlich die Anfangsposition der Teilketten Xkoord, Ykoord und Zkoord in Variablen Anf_X, Anf_Y, Anf_Z festgehalten werden.

```
Steuerung: TASK ;
    ...
    DCL   (Anf_X, Anf_Y, Anf_Z)   INV   FIXED   INIT (1, 9, 13) ;
    ...
    Xkoord := Rfz_Zustand.BIT ( Anf_X : Anf_X + 7 ) ;
    Ykoord := Rfz_Zustand.BIT ( Anf_Y : Anf_Y + 3 ) ;
    Zkoord := Rfz_Zustand.BIT ( Anf_Z ) ;
    ...
END ; ! Steuerung
```

Es sind auch Zuweisungen an Kettenausschnitte möglich; außerdem dürfen sie als Operanden in Ausdrücken auftreten.

Teile eines Ausdrucks können geklammert werden, um die Reihenfolge der Ausdrucksberechnung (vgl. 6.3) zu beeinflussen, z.B.

```
A - ( B + C )
```

6.1.1 Monadische Operatoren

Die folgende Tabelle beschreibt für jeden aufgeführten monadischen Operator

- welchen Typ der Operand haben darf
- welchen Typ des Ergebnis (der Operation) hat und
- die Bedeutung des Operators.

Dabei steht a für irgendeinen Operanden, g für die Genauigkeit, lg für die Länge des Operanden und e für das Ergebnis der Operation.

Syntax	Typ des Operanden a	Typ des Ergebnisses e	Bedeutung
+a	FIXED (g) FLOAT (g) DURATION	FIXED (g) FLOAT (g) DURATION	keine
-a	FIXED (g) FLOAT (g) DURATION	FIXED (g) FLOAT (g) DURATION	Umkehrung des Vorzeichens von a
NOT a	BIT (lg)	BIT (lg)	Umkehrung aller Bitstellen von a

Beispiel:

```
DCL    (X, Y)  FLOAT,
       B       BIT (4) ;
X := 3 ;
B := '1001'B
Y := -X ;              /* Y hat den Wert -3 */
B := NOT B ;           /* B hat den Wert '0110'B */
```

Syntax	Typ des Operanden a	Typ des Ergebnisses e	Bedeutung
ABS a	FIXED (g) FLOAT (g) DURATION	FIXED (g) FLOAT (g) DURATION	e:= | a | (Absolutbetrag von a)
SIGN a	FIXED (g) FLOAT (g) DURATION	FIXED (1)	e := + 1 für a > 0 0 für a = 0 - 1 für a < 0
LWB a	Feld	FIXED (31)	e := untere Grenze der ersten Dimension von a
UPB a	Feld	FIXED (31)	e := obere Grenze der ersten Dimension von a
	CHAR (lg)	FIXED (15)	e := lg

Im allgemeinen müssen bei einer Zuweisung der Typ der links vom Zuweisungs-zeichen angegebenen Variablen und der Typ des Wertes des zugewiesenen Ausdrucks übereinstimmen (vgl. Punkt 6.3). Insbesondere dürfen an Variablen für ganze Zahlen nur Werte vom Typ ganze-Zahl zugewiesen werden. Außerdem müssen die Typen der Operanden von dyadischen Operatoren verträglich sein (vgl. 6.1.2).

Deshalb stehen einige Operatoren zur Verfügung, die eine gegebenenfalls nötige Wandlung des Typs von Objekten bewirken.

Die folgende Tabelle beschreibt für jeden aufgeführten monadischen Typwandlungsoperator

- welchen Typ der Operand haben darf
- welchen Typ das Ergebnis (der Operation) hat und
- die Bedeutung des Operators.

Dabei steht a für irgendeinen Operanden, e für das Ergebnis, g für die Genauigkeit und lg für die Länge des Operanden und des Ergebnisses.

Syntax	**Typ des Operanden a**	**Typ des Ergebnisses e**	**Bedeutung**
TOFIXED a	CHARACTER (1)	FIXED (8)	e hat als Wert die ganze Zahl, die dem Zeichen laut ASCII-Code zugeordnet ist.
	BIT (lg)	FIXED (g)	e hat als Wert die Interpretation des Bitmusters von a als ganze Zahl mit g = lg.
TOFLOAT a	FIXED (g)	FLOAT (g)	e hat als Wert die a entsprechende Gleitpunktzahl.
TOBIT a	FIXED (g)	BIT (lg)	e hat als Wert die Interpretation des Bitmusters von a als Bitkette mit lg = g.
TOCHAR a	FIXED	CHARACTER (1)	e hat als Wert das Zeichen, das der ganzen Zahl laut ASCII-Code zugeordnet ist.
ENTIER a	FLOAT (g)	FIXED (g)	e = größte ganze Zahl kleiner gleich a.
ROUND a	FLOAT (g)	FIXED (g)	e = nächste ganze Zahl gemäß DIN.

Beispiele:

```
DCL   A   FIXED (15),
      X   FIXED (31),
      (B, C)   BIT (15) ;
...
A := ENTIER X // 2 ;      /* Entspricht A := (Entier X)//2 ; */
A := ENTIER (X / 2.0) ;
C := TOBIT A   AND   B ;
A := ROUND X   +   TOFIXED   B;
```

Syntax	Typ des Operanden a	Typ des Ergebnisses e	Bedeutung des Ergebnisses e
SQRT a			Wurzel von a
SIN a			Sinus von a
COS a			Cosinus von a
EXP a	FIXED(g)	FLOAT(g)	e^x (e = 2.718281828459)
LN a	FLOAT(g)	FLOAT(g)	y mit e^y = a
TAN a			Tangens von a
ATAN a			Arcus tangens von a
TANH a			Tangens hyperbolicus von a

Syntax	Ergebnistyp	Bedeutung des Ergebnisses
SIZEOF Bezeichner	FIXED(31)	Speichergröße des Objektes zur Laufzeit in Bytes
SIZEOF einfacher-Typ	FIXED(31)	(wird nicht durch den PEARL-Compiler ausgewertet)

6.1.2 Dyadische Operatoren

Die folgende Tabelle beschreibt für jeden aufgeführten dyadischen Operator

- welchen Typ die Operanden haben dürfen
- welchen Typ das Ergebnis (der Operation) besitzt   und
- die Bedeutung des Operators.

Dabei bezeichnen

- a bzw. b den ersten bzw. zweiten Operanden
- g1, g2, ... bzw. lg1, lg2, ... die Genauigkeiten bzw. Längen der Operanden und des Ergebnisses.

Syntax	Typ von a	Typ von b	Typ des Ergebnisses	Bedeutung
a + b	FIXED(g1)	FIXED(g2)	FIXED(g3)	Addition der Werte von a
	FIXED(g1)	FLOAT(g2)	FLOAT(g3)	und b.
	FLOAT(g1)	FIXED(g2)	FLOAT(g3)	g3 = max (g1, g2)
	FLOAT(g1)	FLOAT(g2)	FLOAT(g3)	
	DURATION	DURATION	DURATION	
	DURATION	CLOCK	CLOCK	
	CLOCK	DURATION	CLOCK	
a - b	FIXED(g1)	FIXED(g2)	FIXED(g3)	Subtraktion der Werte von
	FIXED(g1)	FLOAT(g2)	FLOAT(g3)	a und b.
	FLOAT(g1)	FIXED(g2)	FLOAT(g3)	g3 = max (g1, g2)
	FLOAT(g1)	FLOAT(g2)	FLOAT(g3)	
	DURATION	DURATION	DURATION	
	CLOCK	DURATION	CLOCK	
	CLOCK	CLOCK	DURATION	
a * b	FIXED(g1)	FIXED(g2)	FIXED(g3)	Multiplikation der Werte
	FIXED(g1)	FLOAT(g2)	FLOAT(g3)	von a und b.
	FLOAT(g1)	FIXED(g2)	FLOAT(g3)	g3 = max (g1, g2)
	FLOAT(g1)	FLOAT(g2)	FLOAT(g3)	
	FIXED(g1)	DURATION	DURATION	
	DURATION	FIXED(g2)	DURATION	
	FLOAT(g1)	DURATION	DURATION	
	DURATION	FLOAT(g2)	DURATION	
a / b	FIXED(g1)	FIXED(g2)	FLOAT(g3)	Division der Werte von
	FLOAT(g1)	FIXED(g2)	FLOAT(g3)	a und b,
	FIXED(g1)	FLOAT(g2)	FLOAT(g3)	falls b ≠ 0 ist.
	FLOAT(g1)	FLOAT(g2)	FLOAT(g3)	
	DURATION	FIXED(g2)	DURATION	g3 = max (g1, g2)
	DURATION	FLOAT(g2)	DURATION	g4 = 31 (implementations-
	DURATION	DURATION	FLOAT(g4)	abhängige Zahl)

Syntax	Typ von a	Typ von b	Typ des Ergebnisses	Bedeutung
a // b	FIXED(g1)	FIXED(g2)	FIXED(g3)	Ganzzahlige Division der Werte von a und b, d.h. das Ergebnis besteht nur aus dem ganzzahligen Anteil. g3 = max (g1, g2)
a REM b	FIXED(g1)	FIXED(g2)	FIXED(g3)	Rest nach Division des ersten Operanden durch zweiten (Modulo-Berech-nung). Es gilt: a == (a // b) * b + a REM b
a ** b	FIXED(g1) FLOAT(g1)	FIXED(g2) FIXED(g2)	FIXED(g1) FLOAT(g1)	Potenzierung der Werte von a und b: a^b
a < b oder a LT b	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) CLOCK DURATION CHAR(lg1)	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) CLOCK DURATION CHAR(lg2)	BIT(1)	Vergleich auf "kleiner als": Ist der Wert von a kleiner als der Wert von b, so hat das Ergebnis den Wert '1'B, anderenfalls '0'B. Zeichenkettenvergleich (Algorithmus siehe unten)
a > b oder a GT b	siehe a < b	siehe a < b	BIT(1)	Vergleich auf "größer als": Ist der Wert von a größer als der Wert von b, so hat das Ergebnis den Wert '1'B. anderenfalls '0'B.
a <= b oder a LE b	siehe a < b	siehe a < b	BIT(1)	Vergleich auf "kleiner oder gleich als": Ist der Wert von a kleiner oder gleich als der Wert von b, so hat das Ergebnis den Wert '1'B, anderenfalls '0'B.

Syntax	Typ von a	Typ von b	Typ des Ergebnisses	Bedeutung
a >= b oder a GE b	siehe a < b	siehe a < b	BIT(1)	Vergleich auf "größer oder gleich als": Ist der Wert von a größer oder gleich als der Wert von b, so hat das Ergebnis den Wert '1'B, anderenfalls '0'B.
a == b oder a EQ b	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) CLOCK DURATION CHAR(lg1) BIT(lg1)	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) CLOCK DURATION CHAR(lg2) BIT(lg2)	BIT(1)	Vergleich auf "gleich": Ist der Wert von a gleich dem Wert von b, so hat das Ergebnis den Wert '1'B. anderenfalls '0'B. Ist lg2 ≠ lg1, so wird die kürzere Zeichen- bzw. Bitkette rechts um | lg2 - lg1 | Leerzeichen bzw. Nullen verlängert.
a /= b oder a NE b	FIXED(g1) FIXED(g1) FLOAT(g1) FLOAT(g1) CLOCK DURATION CHAR(lg1) BIT(lg1)	FIXED(g2) FLOAT(g2) FIXED(g2) FLOAT(g2) CLOCK DURATION CHAR(lg2) BIT(lg2)	BIT(1)	Vergleich auf "ungleich": Ist der Wert von a ungleich dem Wert von b, so hat das Ergebnis den Wert '1'B, anderenfalls '0'B. Zeichen- bzw. Bitketten werden rechts ggfs. mit Leerzeichen bzw. Nullen aufgefüllt.

Damit das Ergebnis einer Vergleichsoperation zweier Zeichenketten vom Anwender nachvollzogen werden kann, wird hier der Vergleichsalgorithmus angegeben (die kürzere Zeichenkette wird dabei rechts mit Leerzeichen aufgefüllt).

Algorithmus zum Vergleich zweier Zeichenketten, dabei gilt "lg3 = max (lg1, lg2)":

```
TYPE Urteil FIXED; DCL (kleiner, gleich, groesser) INV FIXED INIT (-1, 0, 1);
    string_vergleich : PROC ( ( string1, string2 ) REF INV CHAR( lg3 ) ) RETURNS (  Urteil  ) ;
        FOR  i  TO  lg3  REPEAT
            IF  TOFIXED  string1.CHAR( i )  <  TOFIXED  string2.CHAR( i )  THEN
                RETURN (  kleiner  ) ;
            ELSE IF  TOFIXED  string1.CHAR( i )  >  TOFIXED  string2.CHAR( i )  THEN
                RETURN (  groesser  ) ;
            FIN ;  FIN ;
        END ; ! Schleife
        RETURN ( gleich ) ;
    END ; ! PROC string_vergleich
```

Syntax	Typ von a	Typ von b	Typ des Ergebnisses	Bedeutung
a AND b	BIT(lg1)	BIT(lg2)	BIT(lg3)	lg3 = max (lg1, lg2). Der kürzere Operand wird rechts um **|** lg2 - lg1 **|** Nullen verlängert. Die folgende Tabelle zeigt den Wert der i-ten Bitstelle des Ergebnisses in Abhängigkeit des Wertes der i-ten Bitstelle von a und b: a b a AND b a OR b a EXOR b 1 1 1 1 0 1 0 0 1 1 0 1 0 1 1 0 0 0 0 0
a OR b	BIT(lg1)	BIT(lg2)	BIT(lg3)	
a EXOR b	BIT(lg1)	BIT(lg2)	BIT(lg3)	
a >< b oder a CAT b	CHAR(lg1) BIT(lg1)	CHAR(lg2) BIT(lg2)	CHAR(lg3) BIT(lg3)	Verkettung: Das Ergebnis ist eine Zeichen- bzw. Bitkette der Länge lg3=lg1+lg2, die aus der Kette a, gefolgt von der Kette b, besteht.
a <> b oder a CSHIFT b	BIT(lg)	FIXED(g)	BIT(lg)	Zyklische Verschiebung von a um b Stellen nach links (b > 0) bzw. rechts (b < 0).
a SHIFT b	BIT(lg)	FIXED(g)	BIT(lg)	a wird um **|** b **|** Stellen nach links (b > 0) bzw. rechts (b < 0) verschoben, wobei Nullen nachgezogen werden.

Da die Genauigkeit des Ergebnisses einer Addition, Subtraktion, Multiplikation oder Division gleich dem Maximum der Genauigkeiten der beiden Operanden ist, kann es vorkommen, daß bei diesen Operationen entstehende Überläufe abgeschnitten werden. Deshalb steht der dyadische Operator FIT zur Verfügung:

Syntax	**Typ von a**	**Typ von b**	**Typ des Ergebnisses**
a FIT b	FIXED (g1) FLOAT(g1)	FIXED(g2) FLOAT(g2)	FIXED(g2) FLOAT(g2)

Er bewirkt, daß die Genauigkeit des Operanden a in die Genauigkeit des Operanden b gewandelt wird. Er hat den Rang 1.

Beispiel:

```
...
DCL     (A, B)   FIXED   (15),
        C   FIXED (31) ;
DCL     D   BIT (16) ;
A := 32767 ;
B := 4 ;
C := (A  FIT  C) * B ;      /* C erhaelt den Wert 131068 */
A := A FIT C ;              /* Zuweisung nicht erlaubt */
A := C FIT A ;              /* OK, aber Informationsverlust von C durch C FIT A */

/* Wandlungen von FIXED- und BIT-Objekten */
/* ohne Änderung der internen Darstellung */
A := (TOFIXED D) FIT A ;
D := TOBIT ( A FIT 1(16) ) ;
```

Weitere dyadische Standard-Operatoren:

Syntax	**Typ des Operanden a**	**Typ des Operanden b**	**Typ des Ergebnisses e**	**Bedeutung**
a LWB b	FIXED (g)	Feld	FIXED(31)	e := untere Grenze der a-ten Dimension von b, wenn diese existiert.
a UPB b	FIXED (g)	Feld	FIXED(31)	e := obere Grenze der a-ten Dimension von b, wenn diese existiert.

Beispiel:

```
...
P :   PROC ( A (,)   FIXED   IDENT ) ;
        ...
        FOR  i  FROM  LWB A  TO  UPB A
                REPEAT
                        FOR  k  FROM  2 LWB A  TO  2 UPB A
                                REPEAT
                                        ...
                                END ; ! for k
                ...
                END ; ! for i
        ...
        END ; ! proc p
...
DCL     Tab1 (5,10)   FIXED,
        Tab2 (-1:2, 3:5)   FIXED ;
...
CALL P (Tab1) ;
...
CALL P (Tab2) ;
```

6.1.3 Berechnung von Ausdrücken

Im folgenden bezeichnen Kleinbuchstaben a, b, c Konstanten oder skalare Variablen.

Gemäß den Regeln der Arithmetik hängt die Reihenfolge der Berechnung eines Ausdruckes davon ab, welchen (Vor-)Rang die einzelnen Operatoren des Ausdrucks besitzen. Der dyadische Operator "*" hat z.B. eine höheren Rang als der dyadische Operator "+": Bei dem Ausdruck "a+b*c" wird also zuerst "b*c" berechnet und dann dieses Produkt zu a addiert.

Die folgende Aufstellung definiert die Rangordnung für dyadische Operatoren, wobei eine niedrigere Zahl einen höheren Rang bedeutet.

Rang	dyadische Operatoren	Auswertungsreihenfolge
1	** , FIT, LWB, UPB	von rechts nach links
2	* , / , >< , //, REM	von links nach rechts
3	+ , - , <> , SHIFT	von links nach rechts
4	< , > , <= , >=	von links nach rechts
5	== , /= , IS , ISNT	von links nach rechts
6	AND	von links nach rechts
7	OR , EXOR	von links nach rechts

Alle monadischen Standard-Operatoren haben den Rang 1.

Die Reihenfolge der Berechnung eines Ausdrucks wird außerdem in der üblichen Weise durch die Klammerung von Teilen des Ausdrucks beeinflußt; z.B. wird bei dem Ausdruck

```
a * ( b - (c-d) )
```

zuerst c-d berechnet, dann dieses erste Zwischenergebnis von b substrahiert und erst dann dieses zweite Zwischenergebnis mit a multipliziert.

Allgemein erfolgt die Berechnung eines Ausdrucks nach folgenden Regeln:

- Der Teilausdruck mit dem ranghöchsten Operator wird zuerst berechnet, sofern hierdurch nicht eine der beiden folgenden Regeln verletzt wird.
- Treten mehrere Operatoren des gleichen Rangs auf, so erfolgt die Berechnung
  - im Fall $2 \geq$ Rang ≥ 7 von links nach rechts
  - im Fall Rang = 1 von rechts nach links
    Beispiel: -a**b entspricht -(a**b)
- Geklammerte Teilausdrücke werden nach obigen Regeln vollständig berechnet, bevor sie mit einem anderen Teilausdruck verknüpft werden.

Beispiel:

```
( a * ( b - c ) ) ** d  <  e + f(x)   AND   h / ( i + j )  >=  ( k - l ) * m
```

Die Striche zeigen, welche Teilausdrücke bei der Berechnung gebildet und mit anderen Teilausdrücken verknüpft werden.

6.2 Operatorvereinbarung (OPERATOR)

Die Operator-Vereinbarung erlaubt, neue Operatoren mit frei wählbaren Bezeichnern zu definieren oder die Bedeutung der bisher voreingestellten Standard-Operatoren zu erweitern.

```
Operator-Vereinbarung ::=
        OPERATOR  Op-Name  ( [ Op-Parameter, ] Op-Parameter)
        RETURNS  (Resultat-Attribut) ;
        Prozedurkörper
        END ;

Op-Name ::=
        Bezeichner | + | - | * | ** | / | // | == | /= | <= | >= | < | > | <> | ><

Op-Parameter ::=
        Bezeichner [ virt-Dimensionsliste ] Parameter-Typ [ IDENTICAL | IDENT ]

Resultat-Attribut ::=
        einfacher-Typ | Typ-Referenz
```

Die Bedeutung ist analog einer Funktionsprozedur (vgl. 8).

Beispiel:

Der Standardoperator + soll für komplexe Zahlen erweitert werden.

```
PROBLEM ;

TYPE   Complex   STRUCT
       [  Real  FLOAT,  Imag  FLOAT  ] ;

OPERATOR  +  (  A  Complex  IDENT ,  B  Complex  IDENT )  RETURNS ( Complex ) ;
       DCL  Sum  Complex ;
       Sum.Real := A.Real + B.Real ;
       Sum.Imag := A.Imag + B.Imag ;
       RETURN ( Sum ) ;
       END ; ! Operator +

DCL    (XX, YY, ZZ)  Complex ,
       (X, Y, Z)     FLOAT ;
. . .
ZZ := XX + YY ;
Z := X + Y ;
. . .
```

Dieses Beispiel zeigt die Möglichkeit, verschiedene Operationen mit demselben Operatornamen zu definieren, wenn die Operanden unterschiedlichen Typs sind. Wenn ein Ausdruck, in dem solch ein Operatorname vorkommt, ausgewertet wird, wird die Operation ausgeführt, bei der die Parametertypen identisch zu den Typen der Operanden in dem Ausdruck sind.

Dient eine Operator-Vereinbarung zur Erweiterung der Bedeutung eines Standard-Operators, so hat der neu vereinbarte Operator den Rang des Standard-Operators. Für einen Operator, der mit einem neuen, nicht standardmäßig eingeführten Operatornamen vereinbart wird, kann mittels einer Rang-Vereinbarung ein Rang zwischen 1 und 7 festgelegt werden:

Rang-Vereinbarung ::=
    PRECEDENCE Op-Name ({ 1 | 2 | 3 | 4 | 5 | 6 | 7 }) ;

Soll für einen neuen Operator ein Rang vereinbart werden, muß dies vor der Operator-Vereinbarung erfolgen; ist keine Rang-Vereinbarung angegeben, erhält der neue Operator den Rang 7.

Beispiel:

...
PRECEDENCE INDEX (1) ;
OPERATOR INDEX ... ;

6.3 Zuweisungen

Zuweisungen sind für skalare Variablen und für Strukturen implementiert, nicht jedoch für Felder.

Zuweisung ::=
    skalare-Zuweisung | Strukturzuweisung

6.3.1 Zuweisungen für skalare Variablen

Zuweisungen für skalare Variablen sind wie folgt definiert:

skalare-Zuweisung ::=
    Name§skalare-Variable { := | = } Ausdruck ;

Die Anweisung bewirkt, daß der links vom Zuweisungszeichen (":=" oder "= ") angegebenen Variablen der Wert des rechts stehenden Ausdrucks zugewiesen wird, d.h. nach der Ausführung der Zuweisung kann man sich mit diesem Namen auf den durch Ausdruck bestimmten Wert beziehen, der gegebenenfalls vor der eigentlichen Zuweisung berechnet wird:

```
Ergebnis (i)  :=   Koeff * SIN (  ( X(i+1)  -  X(i) )  /  X(i)  ) ;
```

Der Typ der links vom Zuweisungszeichen angegebenen Variablen und der Typ des Wertes des Ausdrucks müssen übereinstimmen mit folgenden Ausnahmen:

- Einer FLOAT-Variablen darf der Wert einer FIXED-Variablen bzw. eine ganze Zahl zugewiesen werden.
- Die Genauigkeit einer links vom Zuweisungszeichen stehenden Zahlen-Variablen darf größer sein als die Genauigkeit des Wertes des Ausdrucks.
- Eine links stehende Bit- bzw. Zeichenkette darf eine größere Länge als der zuzuweisende Wert haben; dieser wird gegebenenfalls rechts mit Nullen bzw. Leerzeichen ergänzt.

Operatoren für nötige Typwandlungen sind in 6.1.2 beschrieben.

Beispiele:

```
DCL    (I, J)   FIXED(15),
       K   FIXED(31),
       (X, Y)   FLOAT,
       Bit8   BIT(8),
       Bit12   BIT(12),
       Text4   CHAR(4),
       Text10   CHAR(10),
       Dauer(2)   DURATION,
       Zeit(2)  CLOCK ;

I := 2.0 ;                      ! Falsch
J := 3 ;
X := J+5 ;  Y := 0;
K := J ;
J := K ;                        ! Falsch
Text10 := 'Ergebnis' ;          ! Text10 hat den Wert 'Ergebnis __'
Bit8 := 'A9F'B4 ;               ! Falsch, weil zu lang
Dauer(1) := 1 HRS ;
Dauer(2) := 30 MIN ;
Zeit(1) := 11:00:00 ;
Zeit(2) := Zeit(1) + (IF   Zeit(1) < 12:00:00
                          THEN   Dauer (1)
                          ELSE   Dauer (2)
                      FIN) ;

Bit8 := '10001100'B ;
Bit12 := Bit8  ><  '11' ;       /* Bit 12 hat den Wert '100011001100'B */
Bit8 := Bit8  CSHIFT  3 ;       /* Bit8 hat den Wert '01100100'B */
Bit12 := Bit12  SHIFT  -6 ;     /* Bit12 hat den Wert '000000100011'B */
```

Es besteht die Möglichkeit, Variablen mit einem Attribut für Zuweisungsschutz (vgl. 5.15) zu vereinbaren. Zuweisungen an solche Variablen führen zu Fehlermeldungen.

6.3.2 Zuweisungen für Strukturen

Einer Struktur können die Werte aller Komponenten einer zweiten Struktur in einer einzigen Anweisung zugewiesen werden:

```
Strukturzuweisung ::=
        Name§Struktur_1   { := | = }   Ausdruck§Struktur_2 ;
```

Dabei erhalten die Komponenten der Struktur_1 die Werte der entsprechenden Komponenten der Struktur_2 zugewiesen. Die beiden Strukturen müssen den gleichen Typ besitzen; d.h. die Anzahl der Komponenten und deren Typen müssen übereinstimmen; eine implizite Typanpassung wie bei skalaren Variablen findet nicht statt.

Beispiel:

```
TYPE    Typ_Messung   STRUCT
     [  Zeit   Zeitstempel,
        Wert   FLOAT(53) ] ;

DCL     Werkstueck   STRUCT
     [  Ident   CHAR (8),
        Qualitaet   Typ_Messung,
        ... ] ;

DCL     Messung  Typ_Messung ;
...
Werkstueck.Qualitaet  :=  Messung ;
```

6.4 Überlagerung von Datenstrukturen

Um Programmierfehler weitgehend auszuschließen, prüft der Compiler die typgerechte Verwendung von Variablen und Werten.

Bei Zuweisungen müssen die Typen der linken und rechten Seite verträglich sein (siehe 6.3: Zuweisungen), und beim Aufruf einer Prozedur müssen die aktuellen Parameter mit den formalen Parametern typverträglich sein (siehe 8.2: Aufruf von Prozeduren).

Es gibt aber immer wieder Situationen, in denen die geforderte Typverträglichkeit ("strong typing") hinderlich ist. Als Beispiel kann hier ein Datenbankinterface genannt werden: die Datenbank soll Anwenderdaten unterschiedlicher Typen mit unterschiedlichen Längen speichern. Dazu benötigen die Routinen nur die Adresse und die Länge der zu speichernden Sätze. Bei strenger Typprüfung müßte für jede Anwendung ein neues Datenbankinterface (mit den gewünschten Anwendertypen) programmiert werden.

PEARL 90 bietet zwei verschiedene syntaktische Ausprägungen für die Überlagerung unterschiedlicher Datentypen. Beide Formen erlauben ausschließlich die Typumwandlung von Adressen der Objekte bei der Zuweisung bzw. Identifizierung. Eine Datenkonvertierung findet in keinem Fall statt. Für die Wandlung von Basistypen gibt es eine ausreichende Anzahl von Standardoperatoren in PEARL (siehe 6.1: monadische und dyadische Operatoren).

6.4.1 Der "BY TYPE"-Operator

Mit dem "BY TYPE"-Operator kann der Typ einer Variablen-Adresse in einen beliebigen anderen Zeigertyp gewandelt werden.

Typwandlungs-Ausdruck ::=
    Name **BY TYPE** Typ

Der Typ der Variablen "Name" wird durch den Operator "BY TYPE" in den angegebenen Typ gewandelt. Das Ergebnis des Typ-Wandlungs-Ausdrucks ist die Adresse von "Name" mit dem Typ "Typ". Diese Typwandlung ist eine Compiler-interne Aktion, die eine Fehlermeldung des Compilers verhindert. Zur Laufzeit wird keine Aktion ausgeführt; insbesondere bleibt der Inhalt von "Name" unverändert.

Beispiel:

Überlagerung von Datenobjekten bei der Zuweisung

```
DCL  var        TYP_A ;
DCL  ptr   REF  TYP_B ;
. . .
ptr := var  BY TYPE  TYP_B ;
```

Die Adresse der Variablen "var" erhält den Typ "TYP_B"; damit ist die Zuweisung an den Zeiger "ptr" korrekt. Über den Zeiger "ptr" kann jetzt auf ein Datenobjekt vom Typ "TYP_B" zugegriffen werden, das die Variable "var" überlagert.

Beispiel:

Überlagerung von Datenobjekten beim Prozeduraufruf

```
SPC p1  ENTRY ( REF TYP_B ) GLOBAL;
SPC p2  ENTRY ( TYP_B IDENT ) GLOBAL;
DCL var TYP_A;
. . .
CALL p1 ( var  BY TYPE  TYP_B );
CALL p2 ( var  BY TYPE  TYP_B );
```

Der Typ des formalen Parameters "var" wird durch den Operator "BY TYPE" in den Typ "TYP_B" gewandelt. In beiden Fällen wird die Adresse der Variablen "var" als Zeiger auf ein Objekt vom Typ "TYP_B" an die Prozeduren übergeben. Beide Aufrufe sind somit korrekt.

Durch das Abschalten der Typ-Kontrolle begibt sich der Programmierer in einen Bereich, in dem ihn der Compiler auf keine Fehler mehr hinweisen kann. In keinem Fall darf mit Hilfe einer überlagerten Datenstruktur über das Ende des zugrundeliegenden Speicherbereichs hinaus gelesen oder geschrieben werden, da es sonst zu schwerwiegenden Fehlern beim Ablauf des Programmes kommen kann (dabei ist ein Datenverlust noch das kleinste Problem).

Der Programmierer sollte auch keinerlei Annahmen über die Ablage der Daten innerhalb des überlagerten Speicherbereichs ausnutzen. Die Ablage der Daten ist i.a. abhängig von der Zielmaschine. Programme, die eine Datenkonvertierung mit Hilfe von überlagerten Datenstrukturen vornehmen, sind hardwareabhängig und nur noch mit großem Aufwand portierbar. Der Zugriff auf die Daten sollte sowohl beim Schreiben als auch beim späteren Lesen immer über dieselbe Datenstruktur ausgeführt werden. Da in diesem Fall die Daten immer typgerecht bearbeitet werden, sind solche Programme portabel, wenn der Platzbedarf der überlagernden Datenstruktur nicht über die platzspendende unterlagerte Variable hinausgeht.

6.4.2 Der "VOID"-Datentyp

Im Gegensatz zu der aktiven Typwandlung durch den "BY TYPE"-Operator wird durch den "VOID"-Datentyp eine passive Möglichkeit zur Überlagerung unterschiedlicher Datenobjekte geboten. Ähnlich der Konstanten "NIL", die an Zeiger-Variablen beliebiger Datentypen zugewiesen werden kann, akzeptiert der Compiler die Zuweisung beliebiger Datenadressen an eine Zeiger-Variable vom Typ "VOID".

Der "VOID"-Datentyp wird als Strukturbeschreibung ohne Komponenten geschrieben und darf nur in Verbindung mit REF benutzt werden.

```
VOID-Datentyp ::=
        STRUCT [ ]
```

Eine Referenzvariable mit diesem Typ kann eine beliebige Variablen-Adresse aufnehmen, jedoch kann die Variable nicht direkt verändert werden. Es muß zunächst die Adresse an eine Referenzvariable mit dem benötigten Typ zugewiesen werden, bevor über diese das Datenobjekt manipuliert werden kann.

Beispiel:

```
Allgemeine Speicherverwaltungsroutine

DCL c_max_puffer                INV   FIXED(31)   INIT ( 10000 );
DCL speicher_block  ( c_max_puffer )  CHAR(1);
DCL frei                              FIXED(31)   INIT ( 0 );
DCL speicher_schutz                   SEMA        PRESET( 1 );


Malloc : PROC ( size FIXED(31) ) RETURNS ( REF STRUCT [ ] ) GLOBAL;
        DCL ptr REF CHAR(1);

        REQUEST speicher_schutz;     /* Zugriff auf globale Variablen synchronisieren */

        IF frei + size >= c_max_puffer THEN
                ptr := NIL;        /* kein Platz mehr */
        ELSE
                ptr := speicher_block( frei + 1 );
                frei := frei + size;
        FIN;

        RELEASE speicher_schutz;
        RETURN ( ptr );
END; ! Malloc
```

```
t : TASK;
        DCL ptr_var_a REF TYP_A;
        DCL ptr_var_b REF TYP_B;

        /* Speicherbereich für verschiedene Typen anfordern */
        ptr_var_a := Malloc ( SIZEOF TYP_A );
        ptr_var_b := Malloc ( SIZEOF TYP_B );
END; ! t
```

Diese zweite Form eignet sich besonders für polymorphe Prozeduren, d.h. Prozeduren, die mit aktuellen Parametern unterschiedlicher Typen aufgerufen werden. Der "VOID"-Datentyp dient zum einen in der Prozedur-Spezifikation als Hinweis, daß die Prozedur mit unterschiedlichen Parameter-Typen arbeitet, zum anderen kann an der Aufrufstelle eine explizite Typwandlung mit dem "BY TYPE"-Operator entfallen.

Beispiel:

Datenbank-Interface

```
SPC lies_satz       ENTRY ( adr REF       STRUCT [ ],  size FIXED(31) ) GLOBAL;
SPC schreib_satz    ENTRY ( adr REF INV   STRUCT [ ],  size FIXED(31) ) GLOBAL;

. . .
DCL anwender_daten TYP_A;

/* Satz mit Anwenderdaten in die Datenbank einfügen */
CALL schreib_satz ( anwender_daten, SIZEOF TYP_A );

/* Anwenderdaten-Satz aus der Datenbank lesen */
CALL lies_satz ( anwender_daten, SIZEOF TYP_A );
```

Die Datenbankroutinen "lies_satz" und "schreib_satz" werden von verschiedenen Anwendungsprogrammen aufgerufen. Sie können Sätze unterschiedlicher Länge speichern und lesen. Dazu benötigen die Datenbankroutinen nur die Adressen und die Längen der Sätze, nicht jedoch die anwendungsspezifischen Datentypen.

7. Anweisungen zur Steuerung des sequentiellen Ablaufs

Eine Task- oder Prozedur-Vereinbarung definiert eine Folge von Anweisungen, die bei der Ausführung der Task oder Prozedur in der definierten Reihenfolge sequentiell abgearbeitet werden, sofern nicht dafür vorgesehene Steueranweisungen die Reihenfolge der Abarbeitung beeinflussen.

Solche Steueranweisungen sind

- die bedingte Anweisung
- die Anweisungsauswahl
- die Leeranweisung
- die Wiederholung
- die Sprung-Anweisung
- die Exit-Anweisung

7.1 Bedingte Anweisung (IF)

Mit Hilfe der bedingten Anweisung wird in Abhängigkeit vom Ergebnis eines Ausdrucks festgelegt, mit welcher Anweisung der Programmablauf fortgesetzt werden soll.

```
bedingte-Anweisung ::=
        IF  Ausdruck  THEN  Anweisung···  [ ELSE  Anweisung ··· ]  FIN ;
```

Das Ergebnis des Ausdrucks muß vom Typ BIT(1) sein. Liefert der Ausdruck den Wert '1'B (wahr), so werden die Anweisungen hinter THEN ausgewertet; anderenfalls werden, sofern angegeben, die Anweisungen hinter ELSE ausgewertet.

Sofern die Ausführung der Anweisungen hinter THEN bzw. ELSE nicht zu einem Sprung aus der bedingten Anweisung führt, wird anschließend die Anweisung hinter FIN ausgewertet.

Beispiel:

```
IF Gradient > Grad_Grenze
        THEN  Alarm;
        ELSE  IF  Gradient> Grad_Schwelle
                        THEN  ALL  1 sec  ACTIVATE  Messung;      ! Haeufiger messen
                FIN;
FIN;
...
```

7.2 Anweisungsauswahl (CASE) und Leeranweisung

Angenommen, eine (Funktions-) Prozedur Steuerung soll zur Steuerung mehrerer gleichartiger Geräte benutzt werden und nach jedem Aufruf eine Zahl zwischen 1 und 4 zurückgegeben mit der Bedeutung:

- Rückgabewert = 1: Auftrag durchgeführt
- Rückgabewert = 2: Auftragsdaten falsch
- Rückgabewert = 3: Gerät nicht ansprechbar
- Rückgabewert = 4: Gerät funktioniert nicht richtig

Die aufgerufene Task Versorgung soll dann je Fall die vorgesehene Maßnahme durchführen.

Zur Programmierung solcher Fallunterscheidungen ist die Anweisungsauswahl geeignet, die in zwei (historisch bedingten) Ausprägungen zur Verfügung steht: Die ältere Ausführung läßt nur ganze Zahlen als Unterscheidungskriterien zu; bei der neueren Ausprägung können auch Zeichen angegeben werden. Zunächst wird die ältere Form beschrieben:

```
Anweisungsauswahl -1 ::=
        CASE  Ausdruck
              { ALT  Anweisung ··· } ···
              [ OUT  Anweisung ··· ]
        FIN;
```

Der Anweisungsfolge hinter dem ersten ALT (Alternative 1) ist die Zahl 1 zugeordnet, der Anweisungsfolge hinter dem zweiten ALT (Alternative 2) die Zahl 2 usw.

Bei Ausführung der Anweisungsauswahl wird der angegebene Ausdruck ausgewertet; er muß einen Wert vom Typ FIXED ergeben. Liegt der ganzzahlige Wert zwischen 1 und der Anzahl der angegebenen Alternativen, so wird die zugeordnete Anweisungsfolge ausgeführt; anderenfalls wird die zugeordnete Anweisungsfolge hinter OUT (sofern angegeben) ausgeführt.

Sofern die ausgewählte Anweisungsfolge keinen Sprung aus der Anweisungsauswahl enthält, wird anschließend die Anweisung hinter FIN ausgewertet.

Beispiel:

Das obige Problem läßt sich folgendermaßen programmieren:

```
Versorgung:   TASK  PRIO 7 ;

Steuerung:    PROC ( Nr  FIXED,           ! Geraet
                    Auftrag   BIT(8)  )   ! Auftragsinf.
              RETURNS  (FIXED);
              ! Prozedurkörper zur Erledigung des Steuerauftrags
              END;   ! Steuerung
              ...
              ! Bildung eines Auftrags für das Gerät mit dem Index Nr

nochmal:      CASE  Steuerung (Nr, Auftrag)
              ALT    ! Auftrag erledigt
                     ;
              ALT    ! Auftragsinf. falsch
                     CALL  Fehler (2);  GOTO  Ende ;
              ALT    ! Geraet tot
                     ACTIVATE  Geraetausfall  PRIO 2;
                     CALL  Fehler(3) ;  GOTO  Ende;
              ALT    ! Geraet fkt. falsch
                     CALL  Geraetkontrolle;  GOTO  nochmal;
              OUT    ! Ergebnis ausser Bereich
                     CALL  Fehler(5);
              FIN;
              ...
Ende:         END;   ! Versorgung
```

In diesem Beispiel wird die Leeranweisung benutzt. Sie besteht nur aus einem Semikolon und hat keine Auswirkungen. Nach dem Schlüsselwort ALT muß mindestens eine Anweisung folgen; dies darf auch die Leeranweisung sein. Sie ist ohne Wirkung und nur in bedingten Anweisungen und Anweisungsauswahlen von Interesse.

In dem Beispiel wird durch die Leeranweisung erreicht, daß im Erfolgsfall ("Auftrag erledigt") direkt die Anweisung hinter FIN ausgeführt wird.

Die allgemeine Form der Leeranweisung lautet:

```
Leeranweisung ::=
        ;
```

Die zweite Ausprägung der Anweisungsauswahl hat folgende Form:

```
Anweisungsauswahl-2 ::=
        CASE  Case-Index
        { ALT  (Case-Liste)  Anweisung ··· } ···
        [ OUT  Anweisung ··· ]
        FIN ;

Case-Index ::=
        Ausdruck§mit-Wert-vom-Typ-FIXED-oder-CHAR(1)

Case-Liste ::=
        Index-Bereich [ , Index-Bereich ] ···

Index-Bereich ::=
        Konstante [ : Konstante ]
```

Alle angegebenen Konstanten müssen vom Typ des Case-Index-Ausdrucks sein; CHAR(1) oder FIXED sind erlaubt.

Bei Ausführung der Anweisungsauswahl-2 wird der Case-Index ausgewertet. Ist der Wert in einer der angegebenen Case-Listen enthalten, so wird die zugeordnete Anweisungsfolge ausgeführt; anderenfalls wird die Anweisungsfolge hinter OUT (sofern angegeben) ausgeführt.

Sofern die ausgewählte Anweisungsfolge keinen Sprung aus der Anweisungsauswahl enthält, wird anschließend die Anweisung hinter FIN ausgewertet.

Beispiele:

```
DCL     (Operator, chr) CHAR(1), (x, y)   FIXED ;
...
CASE    Operator
        ALT     ('+')     x := x + y ;
        ALT     ('-')     x := x - y ;
        ALT     ('*')     x := x * y ;
        ALT     ('/')     CASE y
                          ALT  (0)   CALL   Error ;
                          OUT   x := x//y ;
                          FIN ;
FIN ;

CASE    chr
        ALT     ('A' : 'Z')  CALL  uppercase ;
        ALT     ('a' : 'z')  CALL  lowercase ;
FIN ;

CASE    chr
        ALT     ('A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u')   CALL   Vocal(chr) ;
        ...
FIN ;
```

Eine gemischte Verwendung der beiden Ausprägungen der Anweisungsauswahl ist nicht vorgesehen. So ist beispielsweise nicht korrekt:

```
CASE  ErrNum
      ALT   /* 1 */     CALL  ok ;
      ALT   (0)         CALL  nothing_done;
      ALT   (-99:-1)    CALL  ErrorMsg (ErrNum) ;
      ...
```

Die Auswahl muß deterministisch sein.

Sprünge in eine Anweisungsauswahl hinein sind verboten.

7.3 Wiederholung (FOR - REPEAT)

Häufig muß eine Anweisungsfolge wiederholt ausgeführt werden, wobei sich nur ein Parameter ändert. Beispielsweise sollen verschiedene Geräte überprüft werden; Anz_Geraete sei die Anzahl der Geräte:

```
FOR  i FROM 1  BY  1  TO  Anz-Geraete
REPEAT
      Überprüfung von Geraet(i)
END ;
```

Allgemein sind solche "Programmschleifen" wie folgt aufgebaut:

```
Wiederholung ::=
      [ FOR Bezeichner§Laufvariable ]
      [ FROM Ausdruck§Anfangswert ]
      [ BY Ausdruck§Schrittweite ]
      [ TO Ausdruck§Endwert ]
      [ WHILE Ausdruck§Bedingung ]
      REPEAT
      [ Vereinbarung ] ··· [ Anweisung ] ···
      END  [ Bezeichner§Schleife ] ;
```

Die hinter REPEAT aufgeführten Vereinbarungen und Anweisungen, d.h. der Schleifenkörper, werden so oft durchlaufen, wie dies durch die davor angegebene Vorschrift vorgegeben ist; anschließend wird die hinter END folgende Anweisung ausgeführt. Es ist jedoch auch möglich, den Schleifenkörper vorzeitig durch eine Sprung-Anweisung oder die Exit-Anweisung (vgl. 7.5) zu verlassen. Sprünge in den Schleifenkörper hinein sind nicht zugelassen.

Im Schleifenkörper sind alle Anweisungen zugelassen; insbesondere können also Wiederholungen ineinandergeschachtelt werden:

```
FOR i TO 10
REPEAT
    FOR k TO 10
    REPEAT
        c (i,k):= a (i,k) + b(i,k);
    END ;
END ;
```

Fehlen Anfangswert oder Schrittweite, so werden sie als 1 angenommen. Fehlt der Endwert, kann der Schleifenkörper unbegrenzt oft wiederholt werden.

Die Laufvariable darf weder vereinbart noch verändert werden; sie hat implizit den Typ FIXED. Die Werte der Ausdrücke für Anfangswert, Schrittweite und Endwert müssen vom Typ FIXED, der Wert des Ausdrucks für die Bedingung vom Typ BIT(1) sein.

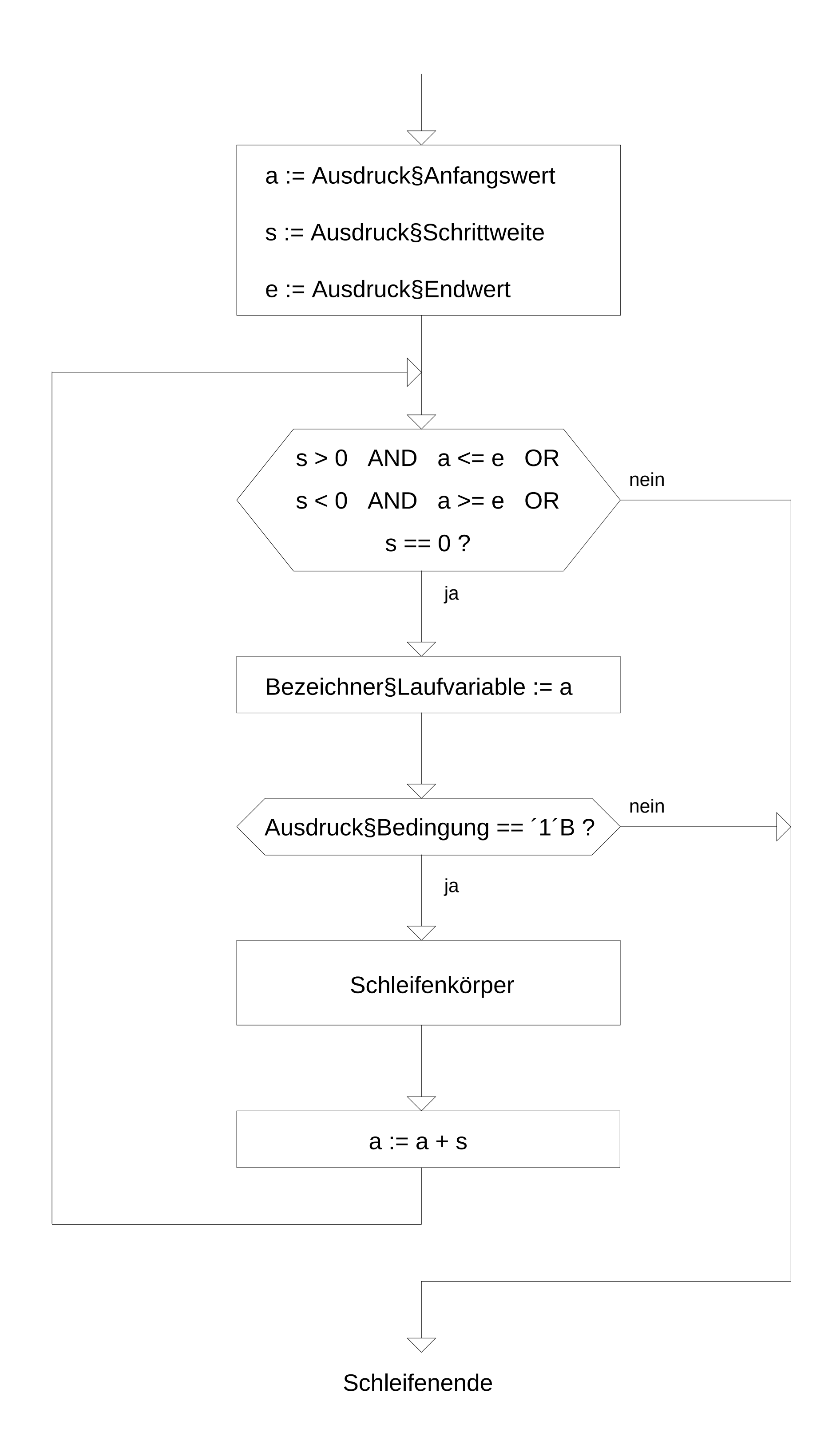
Die Laufvariable darf in den angegebenen Ausdrücken mit Ausnahme von Ausdruck§Bedingung nicht verwendet werden, wohl aber in den zu wiederholenden Anweisungen.

Im übrigen gelten für den Schleifenkörper alle Regeln für Blöcke (vgl. 4.4).

Das folgende Flußdiagramm ist eine äquivalente Darstellung der Anweisung

```
FOR Bezeichner§Laufvariable
FROM Ausdruck§Anfangswert
BY Ausdruck§Schrittweite
TO Ausdruck§Endwert
WHILE Ausdruck§Bedingung
REPEAT
    Schleifenkörper
END ;
```

Bild: Flußdiagramm zur Auswertung einer Wiederholungsanweisung



7.4 Sprung-Anweisung (GOTO)

Sprung-Anweisung ::=
  GOTO Bezeichner§Marke ;

Diese Anweisung bewirkt, daß der Programmablauf an der durch den Markenbezeichner bestimmten Programmstelle fortgesetzt wird. Diese Programmstelle muß eine Anweisung sein und darf nicht außerhalb des Körpers der Task oder Prozedur liegen, die die Sprung-Anweisung ausführt.

Beispiel:

```
                 ...
messen: lesen :  READ Wert FROM Geraet;
                 ...
                 GOTO lesen;
```

Allgemein können Anweisungen (mehrfach) mit Marken markiert werden; d.h. die Marke wird, durch Doppelpunkt getrennt, unmittelbar vor der (gegebenenfalls bereits markierten) Anweisung angegeben.

7.5 Exit-Anweisung (EXIT)

Die Exit-Anweisung dient zum gezielten Verlassen von Blöcken und Schleifen. Mit EXIT können auch gezielt mehrfach geschachtelte Blöcke und Schleifen verlassen werden, die dann am entsprechenden Ende einen Bezeichner (das Sprungziel) tragen müssen.

Exit-Anweisung ::=
  EXIT [Bezeichner§Block-oder-Schleife] ;

Fehlt der Bezeichner, so wird der Programmablauf mit der Anweisung fortgesetzt, die dem Ende des Blocks oder der Schleife folgt, in der die Exit-Anweisung steht.

Ist der Bezeichner angegeben, so wird der Programmablauf mit der Anweisung fortgesetzt, die dem Ende des bezeichneten Blocks oder der bezeichneten Schleife folgt. Dabei muß die Exit-Anweisung in einem inneren Block oder einer inneren Schleife stehen.

Die Exit-Anweisung dient nicht zum Verlassen von Prozeduren oder Tasks.

Beispiel:

```
...
BEGIN                            /* Analyse */
      ...
      TO  Anzahl REPEAT          /* Vergleich */
            ...
            IF   Messwert < Grenzwert
            THEN   EXIT   Analyse ;
            ELSE   ...
            FIN ;
            ...
      END   Vergleich ;
      ...
END   Analyse ;
RETURN   (OK) ;
      ...
```

Der Ausführung von "EXIT Analyse ; " würde umittelbar die Ausführung von "RETURN (OK) ; " folgen.

8. Prozeduren

Bei der Lösung einer Automationsaufgabe formuliert man im Sinne einer strukturierten Programmierung für einen logisch unabhängigen Algorithmus einen selbständigen Programmteil und gibt ihm einen Namen, zumal wenn die Durchführung des Algorithmus an mehreren Stellen des gesamten Programms benötigt wird, wobei sich eventuell nur die Argumente des Algorithmus, seine Parameter, ändern. Die Ausführung eines solchen Programmteils wird durch den Aufruf seines Namens
- gegebenenfalls versehen mit aktuellen Parameterwerten - angestoßen.

Soll dieser Aufruf dieselbe Wirkung haben, als ob an seiner Stelle der aufgerufene Programmteil ausgeführt würde, so wird dieser Programmteil in PEARL als Prozedur vereinbart und aufgerufen. Anderenfalls - wenn nämlich die dem Aufruf folgenden Anweisungen zeitlich parallel zu dem aufgerufenen Programmteil ausgeführt werden soll - vereinbart und startet man den Programmteil als Task. Tasks werden im Abschnitt 9, Parallele Aktivitäten, behandelt.

Prozeduren, die an ihre Aufrufstelle ein Ergebnis zurückgeben, heißen Funktionsprozeduren, alle anderen werden Unterprogramm-Prozeduren genannt.

Beispiel für eine Unterprogramm-Prozedur:

Die Prozedur Ausgabe möge eine Positionsangabe Position vom Typ FIXED in eine Bitkette BinPos wandeln und an eine zu positionierende Maschine ausgeben, die durch die Nummer Masch_Nr vom Typ FIXED gekennzeichnet sei. Ausgabe werde u.a. von der Task Steuerung aufgerufen.

```
PROBLEM ;
     Ausgabe :  PROC  ( ( Position, Masch_Nr )  FIXED  ) ;
        DCL  BinPos  BIT (8) ;
        ! Übertragung von Position in BinPos
        ! Ausgabe von BinPos an die Maschine Masch_Nr
        END ; ! Deklaration von Ausgabe

     Steuerung : TASK ;
        DCL    ( Pos ,      /* Aktuelle Soll-Position */
                 Nr         /* Nr der Maschine */     )  FIXED ;
        ...
        /* Zuweisungen an Pos und Nr */
        CALL Ausgabe (Pos, Nr) ;
        ...
        END ; ! Deklaration von Steuerung
     ...
```

Position und Masch_Nr sind die formalen Parameter von Ausgabe; Pos und Nr sind aktuelle Parameter. Binpos ist eine lokale Variable von Ausgabe, die nur innerhalb von Ausgabe bekannt ist.

Beispiel für eine Funktionsprozedur:

Aufgrund eines Belegungsplans Bel_Plan soll die Prozedur Next_Machine die Nummer der Maschine bestimmen, die unter allen verfügbaren Maschinen als nächste zu belegen ist. Dabei soll Bel_Plan nicht als Parameter übergeben werden; die zurückzugebende Nummer sei vom Typ FIXED. Next_Machine soll innerhalb der Task Versorgung deklariert und aufgerufen werden.

```
PROBLEM ;
     DCL    Bel_Plan   ... ;

     Versorgung: TASK ;
            DCL   Masch_Nr   FIXED ;
            ...
            Next_Machine: PROCEDURE   RETURNS (FIXED) ;
                  DCL   Nr   FIXED ; ! Nr. der naechsten Maschine
                    ! Feststellen von Nr mit Hilfe von Bel_Plan
                  RETURN (Nr) ;
               END ; ! Deklaration von Next_Machine
            ...

            Masch_Nr := Next_Machine ;
            ...
         END ; ! Deklaration von Versorgung
     ...
```

Da die Variable Bel_Plan auf Modulebene vereinbart ist, kann sie von allen Prozeduren und Tasks des Moduls benutzt und ggf. verändert werden.

8.1 Vereinbarung von Prozeduren (PROC)

Die Anweisungsfolge, die beim Aufruf einer Prozedur anstelle des Aufrufs ausgeführt werden soll, wird in einer Prozedur-Deklaration unter Angabe eines Prozedur-Bezeichners festgelegt. Die Anweisungen der Prozedur können Daten benutzen,

- die auf Modulebene oder in einem übergeordneten Programmbereich (siehe 4.1) vereinbart sind,
- die als formale Parameter spezifiziert sind, d.h. als Platzhalter für die Ausdrücke oder Variablen, die der Prozedur beim Aufruf als aktuelle Parameter übergeben werden oder
- die in der Prozedur lokal vereinbart sind.

Die lokalen Vereinbarungen und die Anweisungen der Prozedur bilden den Prozedurkörper.

Prozedur-Deklaration ::=
    Bezeichner : { **PROCEDURE** | **PROC** } [Liste-formaler-Parameter]
    [Resultat-Attribut]
    [Global-Attribut] ;
    Prozedurkörper
    END ;

Prozedurkörper ::=
    [Vereinbarung ···] [Anweisung ···]

Liste-formaler-Parameter ::=
    (Parameterangabe [, Parameterangabe] ···)

Parameterangabe ::=
    Bezeichner-Angabe [virtuelle-Dimensionsliste] [Zuweisungsschutz]
    Parametertyp [**IDENTICAL** | **IDENT**]

virtuelle-Dimensionsliste ::=
    ([, ···])

Parametertyp ::=
    einfacher-Typ | Typ-Referenz | Typ-Struktur
    | Bezeichner§für-Typ | Typ-Dation | Typ-Echtzeit-Objekt

Typ-Echtzeit-Objekt ::=
    SEMA | **BOLT** | **IRPT** | **INTERRUPT** | **SIGNAL**

Resultat-Attribut ::=
    RETURNS (Resultat-Typ)

Resultat-Typ ::=
einfacher-Typ | Typ-Referenz | Typ-Struktur | Bezeichner§für-Typ

Die allgemeine Form der Spezifikation einer Prozedur lautet wie folgt:

Prozedur-Spezifikation ::=
{ **SPECIFY** | **SPC** } Bezeichner§Prozedur { **ENTRY** | [:] **PROC** }
[Liste-der-Parameter-für-SPC] [Resultat-Attribut] Global-Attribut ;

Liste-der-Parameter-für-SPC ::=
(Parameter-Spezifikation [, Parameter-Spezifikation] ···)

Parameter-Spezifikation ::=
[Bezeichner] [virtuelle-Dimensionsliste] [Zuweisungsschutz]
Parametertyp [**IDENTICAL** | **IDENT**]

In der Parameter-Spezifikation hat die (optional) angebbare Liste der Parameter nur einen dokumentarischen Wert; jedoch ist es so möglich, den Kopf einer Prozedurdeklaration in ein anderes Modul zu kopieren und durch Ergänzen des Schlüsselwortes SPECIFY eine korrekte Spezifikation der Prozedur zu erzeugen.

Unterprogramm-Prozeduren werden ohne, Funktionsprozeduren mit Resultat-Attribut vereinbart. Der Resultat-Typ bestimmt den Typ des berechneten Ergebnisses, das an die Aufrufstelle zurückgegeben wird. Diese Rückgabe erfolgt mittels der Return-Anweisung in der Form

RETURN (Ausdruck) ;

Der Wert des Ausdrucks muß also den durch das Resultat-Attribut spezifizierten Typ besitzen.

Die Abarbeitung des Prozedurkörpers einer Funktionsprozedur wird durch Ausführung einer Return-Anweisung beendet. Anders dürfen Funktionsprozeduren nicht verlassen werden.

Die Abarbeitung einer Unterprogramm-Prozedur wird beendet durch

- die Ausführung der Return-Anweisung in der Form

  RETURN ;

- oder die Ausführung der letzten Anweisung des Prozedurkörpers.

Der Prozedurkörper kann Vereinbarungen enthalten, z.B. die Vereinbarungen lokaler Variablen, die dann nur innerhalb des Prozedurkörpers bekannt sind. Es können aber

auch weitere Prozeduren, sogenannte eingeschachtelte Prozeduren, vereinbart werden; die dabei auftretenden Aspekte der Eindeutigkeit von Namen, die auch bei der Vereinbarung von Prozeduren in Taskkörpern auftreten, sind in 4.3 im Zusammenhang mit Blöcken beschrieben.

Durch den Aufruf werden den spezifizierten formalen Parametern der Prozedur als aktuelle Parameter Variable oder Ausdrücke zugeordnet. Auf welche Art (von zwei Möglichkeiten) diese Zuordnung erfolgt, wird dadurch bestimmt, ob das Attribut IDENTICAL angegeben ist oder nicht. Beide Arten werden in 8.2, Aufruf von Prozeduren, erklärt.

Die Anzahl n der Kommata in der virtuellen Dimensionsliste gibt an, daß der Parameter ein (n+1)-dimensionales Feld ist. Formale Feldparameter (virtuelle Dimensionsliste ist vorhanden) dürfen nur zusammen mit dem IDENTICAL-Attribut spezifiziert werden. Soll z.B. das eindimensionale Feld "A(10)FIXED" an eine Prozedur P mit dem entsprechenden formalen Parameter Feld übergeben werden, so ist Feld so zu spezifizieren: "Feld ( ) FIXED IDENTICAL".

Auf Modulebene deklarierte Prozeduren werden vom Compiler *reentrant-fähig* übersetzt, so daß sie von mehreren Tasks (siehe 9) simultan benutzt werden können. Der rekursive Aufruf von Prozeduren ist für alle - auch für die eingeschachtelten - Prozeduren erlaubt. Da für jede Task aber nur ein begrenzter Speicherbereich für die lokalen Daten der aufgerufenen Prozeduren (Stack) zur Verfügung steht, sollte der Programmierer im Sinne von sicheren Programmen die Rekursion vermeiden (oder geeignet begrenzen).

8.2 Aufruf von Prozeduren (CALL)

Unterprogramm-Prozeduren werden mit Hilfe des Schlüsselwortes CALL oder nur mit ihrem Bezeichner aufgerufen:

```
Call-Anweisung ::=
        [ CALL ]   Name§Unterprogramm-Prozedur   [ Liste-aktueller-Parameter ] ;

Liste-aktueller-Parameter ::=
        (  Ausdruck  [ , Ausdruck ] ...  )
```

Beispiel:

```
SPC    Ausgabe  PROC  ( P  FIXED , N  FIXED )  GLOBAL ;
DCL    (Pos, Nr)  FIXED ;
...
! Zuweisungen an Pos und Nr
CALL   Ausgabe (Pos, Nr) ;
```

Die Call-Anweisung bewirkt, daß den formalen Parametern der bezeichneten Prozedur die angegebenen aktuellen Parameter in der Reihenfolge der Niederschrift zugeordnet werden und dann der Prozedurkörper ausgeführt wird. Anschließend wird die auf die Call-Anweisung folgende Anweisung ausgeführt.

Der Aufruf einer Funktionsprozedur erfolgt nicht als selbständige Anweisung, sondern innerhalb von Ausdrücken unter Angabe des Bezeichners und der aktuellen Parameter:

```
Funktionsaufruf ::=
        Name§Funktionsprozedur [ Liste-aktueller-Parameter ]
```

Beispiel:

Die Funktionsprozedur Ari soll das arithmetische Mittel eines Feldes von n FLOAT-Variablen berechnen. Dieses Mittel soll dann zusammen mit dem Text 'Arith.Mittel' ausgedruckt werden.

```
Ari:    PROC  ( Feld ( )  FLOAT  IDENTICAL )  RETURNS  ( FLOAT ) ;
            DCL  Summe  FLOAT ;
            DCL  (UGR, OGR)  FIXED ;
            Summe := 0 ;
            UGR := LWB Feld ;
            OGR := UPB Feld ;
            FOR  i  FROM  UGR  BY  1  TO  OGR
            REPEAT
                Summe := Summe + Feld (i) ;
            END ; ! Schleife
            RETURN  ( SUMME / (OGR - UGR  + 1) ) ;
        END ; ! Ari
```

```
DCL     Messwert(10)   FLOAT ;
...
        /* Erfassen der Messwerte */
PUT     Ari ( Messwert ) ,   'Arith.Mittel'   TO   Drucker   BY   LIST ;
...
```

Bei der Auswertung eines Funktionsaufrufs werden den formalen Parametern der bezeichneten Funktionsprozedur die angegebenen aktuellen Parameter in der Reihenfolge der Niederschrift zugeordnet; sodann wird der Prozedurkörper ausgeführt. Anschließend wird in der Auswertung des Ausdrucks fortgefahren, in dem der Funktionsaufruf erfolgte - in obigem Beispiel also mit der Auswertung des Ausdrucks 'Arith.Mittel' in der Put-Anweisung.

Sowohl bei der Call-Anweisung als auch beim Funktionsaufruf müssen die Typen der aktuellen Parameter den Typen der zugehörigen formalen Parameter entsprechen.

Die Zuordnung der aktuellen Parameter zu den formalen Parametern kann auf zwei Arten erfolgen: Wenn die Spezifikation eines formalen Parameters mit dem Zusatz IDENTICAL oder IDENT versehen ist, geschieht die Zuordnung mittels Identifizierung, anderenfalls durch Wertübertragung.

Im Fall der Wertübertragung (auch *call by value* genannt) wird beim Aufruf der Prozedur für jeden vereinbarten formalen Parameter ein neues Objekt vereinbart, das den Typ des formalen Parameters besitzt und lokal bzgl. des Prozedurkörpers ist; d.h. die formalen Parameter werden lokale Variablen vom spezifizierten Typ. Sodann werden die Werte der aktuellen Parameter den entsprechenden formalen Parametern zugewiesen. Eine Zuweisung an einen formalen Parameter durch eine Anweisung im Prozedurkörper bewirkt also keine Veränderung des aktuellen Parameters. Außerdem dürfen in diesem Fall beliebige Ausdrücke als aktuelle Parameter übergeben werden.

Bei der Zuordnung mittels Identifizierung (auch *call by reference* genannt) wird ein formaler Parameter mit dem entsprechenden aktuellen Parameter identifiziert; d.h. unter dem Namen des formalen Parameters bezieht man sich im Prozedurkörper auf die Daten des aktuellen Parameters. Eine Zuweisung an einen formalen Parameter im Prozedurkörper bedeutet hier also eine Zuweisung an die Variable, die als entsprechender aktueller Parameter übergeben wurde. Deshalb dürfen in diesem Fall keine beliebigen Audrücke, sondern nur Namen (von Variablen) als aktuelle Parameter übergeben werden.

Beispiel:

```
PROBLEM ;

P1:   PROC  ( pi  FIXED , pj  FLOAT  IDENT ) ;
      ...
      pi := 3 ; pj := 5.0 ;
      END ; ! P1

P2:   PROC ... ;
      DCL  (i, j)  FIXED,  a(100)  FLOAT ;
      ...
      i := 2 ;   a(i) := 2.5;
      CALL  P1  ( i,  a(i) ) ;
      ...
      END ; ! P2
...
```

Nach dem Aufruf von P1 in P2 hat i (immer noch) den Wert 2, aber a(i) den Wert 5.0.

Wie bereits die Sprachform der Prozedur-Deklaration (siehe 8.1) zeigt, dürfen die Werte der aktuellen Parameter vom Typ

- ganze-Zahl oder Gleitpunktzahl oder
- Bitkette oder Zeichenkette oder
- Uhrzeit oder Zeitdauer oder
- Struktur oder Bezeichner§für_Typ oder
- Typ-Referenz

sein.

Objekten der Typen

- DATION, SEMA, BOLT und INTERRUPT

sind keine expliziten Werte zugeordnet. Solche Objekte dürfen nur per Identifizierung an eine Prozedur übergeben werden, d.h. der formale Parameter muß mit dem IDENTICAL-Attribut vereinbart worden sein.

8.3 Referenzen auf Prozeduren (REF PROC)

Die Möglichkeit, Prozedur-Referenzvariablen zu nutzen, ist ein erster Schritt Richtung objektorientierte Programmierung. Damit lassen sich z.B. Datenstrukturen und die notwendigen Prozeduren zur kontrollierten Manipulation dieser Strukturen zu neuen abstrakten Datenstrukturen zusammenfassen.

Eine Deklaration von Referenzvariablen für Prozeduren enthält die Beschreibung aller Parametertypen, sowie den Typ des Resultats.

Prozedur-Referenz-Deklaration ::=
    { **DECLARE** | **DCL** }   Bezeichner-Angabe   [Dimensions-Attribut] [**INV**]
    REF   Typ-Prozedur   [Global-Attribut] [Initialisierungsattribut] ;

Typ-Prozedur ::=
    PROC   [Liste-der-Parameter-für-SPC]   [Resultat-Attribut]

Die allgemeine Form der Spezifikation von Prozedur-Referenz-Variablen lautet:

Prozedur-Referenz-Spezifikation ::=
    { **SPECIFY** | **SPC** }   Bezeichner-Angabe   [virtuelle-Dimensionsliste] [**INV**]
    REF   Typ-Prozedur   Global-Attribut ;

Die Wertzuweisung an eine Prozedur-Referenzvariable geschieht mit der Zuweiung:

Zuweisung ::=
    Name§Ref-Proc-Variable   { := | = }   Bezeichner§Prozedur ;

Dabei dürfen nur auf Modulebene deklarierte Prozeduren zugewiesen werden.

Die geforderte Übereinstimmung der Typen bedeutet in diesem Fall, daß die Anzahl der Parameter, sämtliche Parametertypen und auch der Typ der Resultat-Attribute übereinstimmen.

Ein Aufruf einer Prozedur über eine Prozedur-Referenzvariable geschieht einfach durch Angabe der Referenzvariablen gefolgt von der Liste der aktuellen Parameter. Bei parameterlosen Prozeduren kann CALL, oder im Falle von Funktionsprozeduren auch CONT, benutzt werden.

Beispiele:

(1)

```
DCL  ProcZeiger  REF PROC (a FIXED IDENT, b FIXED IDENT, c FIXED);

Addiere:  PROC (a FIXED IDENT, b FIXED IDENT, c FIXED);
          c  :=  a  +  b ;
          END ;

DCL (A, B, C) FIXED;
ProcZeiger := Addiere;
ProcZeiger (A, B, C);
```

(2)

```
DCL  FuncZeiger  REF  PROC  RETURNS(CLOCK);
Uhrzeit:  PROC  RETURNS(CLOCK);
          RETURN(NOW);
          END;

DCL(A, B) CLOCK;
FuncZeiger  :=  Uhrzeit;
A  :=  FuncZeiger ;
B  :=  CONT  FuncZeiger;
```

9. Parallele Aktivitäten

Typisch für ein Programm zur Steuerung eines technisches Prozesses sind

- asynchrone, d.h. zeitlich parallele, voneinander unabhängige Abläufe von Programmteilen, die durch spontane Ereignisse oder zu bestimmten (eingeplanten) Zeiten angestoßen werden, sowie
- die Synchronisation solcher Abläufe an bestimmten Programmstellen, z.B. um Daten untereinander austauschen zu können.

Zur Programmierung solcher Vorgänge werden in PEARL Tasks, Interrupts und Synchronisationsgrößen benutzt.

Eine Task ist der Ablauf eines Programmstücks unter der Kontrolle des Betriebssystems. Dieses Programmstück, der Taskkörper, besteht wie ein Prozedurkörper aus PEARL-Vereinbarungen und -Anweisungen. Vor ihrer Benutzung muß eine Task unter Angabe ihres Körpers vereinbart werden; dabei erhält sie einen Bezeichner zugeordnet, unter dem sie im folgenden beeinflußt wird, z.B. gestartet oder verzögert werden kann.

Da den Tasks eines Programms normalerweise nur ein Prozessor zur Verfügung steht, müssen sie um dessen Benutzung konkurrieren - aber auch um den Zugriff auf andere Betriebsmittel (wie z.B. E/A-Geräte), die gemeinsam benutzt werden müssen. Das Betriebssystem sollte die Betriebsmittel jedoch unter Berücksichtigung der Dringlichkeit der Tasks vergeben. Deshalb kann einer Task eine positive ganze Zahl als Priorität zugeordnet werden, wobei niedrigere Zahlen eine höhere Priorität bedeuten. Die so festgelegten Dringlichkeiten der Tasks werden vom Betriebssystem zur Steuerung der Betriebsmittelvergabe benutzt. Besitzt z.B. eine Task den einzigen Prozessor und fordert dabei ein anderes, exklusiv belegtes Betriebsmittel an, so wird ihr vom Betriebssystem der Prozessor entzogen; von den auf die Benutzung des Prozessors wartenden Tasks erhält sodann diejenige mit der höchsten Priorität den Prozessors zugeteilt. Gleichprioren lauffähigen Tasks wird der Prozessor nach der Round-Robin-Strategie zugeteilt.

Eine solche prioritätsgesteuerte Neuvergabe eines Prozessors findet jedesmal statt, wenn eine seiner Betriebssystemfunktionen angesprochen wird, z.B. bei Eintritt eines Interrupts oder bei der Durchführung von Anweisungen zur Steuerung von Tasks, zur Synchronisierung, zur Ein- und Ausgabe usw.

9.1 Vereinbarung von Tasks (TASK)

Die Vereinbarung von Tasks erfolgt analog zur Deklaration von Prozeduren. Im Gegensatz zu Prozeduren dürfen Tasks jedoch nur auf Modulebene - also nicht innerhalb von Prozedur- oder Taskkörpern - vereinbart werden. Außerdem sind Parameter unzulässig. Da in einem Taskkörper jedoch alle auf Modulebene vereinbarten PEARL-Objekte benutzt und - soweit möglich - verändert werden dürfen, kann der Datenaustausch mit einer Task durch Daten erfolgen, die auf Modulebene vereinbart sind. Insbesondere der Zugriff mehrerer Tasks auf dieselben Daten sollte jedoch mit den in 9.3 beschriebenen Mitteln sorgfältig synchronisiert werden. Der Datenaustausch durch E/A-Anweisungen ist ebenfalls möglich; hier übernehmen die Ein- und Ausgabe-Funktionen bereits die Synchronisierung.

Beispiel:

Eine Task Protokoll hinterlegt Protokolltexte in der Variablen Text, die durch die Task Ausgabe an ein Terminal ausgegeben werden sollen. (Die erforderlichen Synchronisierungsanweisungen werden in 9.3 erläutert.)

```
PROBLEM ;
    DCL  Text  CHAR (60) ;
    Protokoll:    TASK ;
                  Text := Protokolltext (27) ;
                  . . .
                  END ;   ! Protokoll

    Ausgabe:      TASK ;
                  PUT  Text  TO  Drucker  BY  LIST ;
                  . . .
                  END ;   ! Ausgabe
    . . .
```

Die allgemeine Form einer Task-Deklaration lautet:

```
Task-Deklaration ::=
    Bezeichner :  TASK [ Prioritätsangabe ]  [ MAIN ] [ Global-Attribut ] ;
    Taskkörper
    END ;

Prioritätsangabe ::=
    { PRIORITY | PRIO }   ganze-Zahl-ohne-Genauigkeit§größer-Null

Taskkörper ::=
    [ Vereinbarung ··· ]  [ Anweisung ··· ]
```

Wird in der Task-Deklaration keine Priorität angegeben, wird die Priorität 255 angenommen.

Die allgemeine Form einer Task-Spezifikation lautet:

Task-Spezifikation ::=
{ **SPECIFY | SPC** } Bezeichner§Task **TASK** Global-Attribut ;

Die mit MAIN gekennzeichneten Tasks werden beim Programm- oder Systemstart gemäß ihrer Priorität gestartet. Alle MAIN-Tasks müssen in einem Modul deklariert sein.

Das Global-Attribut wird in 4.4 erklärt.

9.1.1 Referenzen auf Tasks (REF TASK)

Die Steuerung von allen Task-Aktivitäten ist auch über Referenzen auf Objekte vom Typ TASK möglich. Die Deklaration von entsprechenden Referenzvariablen kann so geschehen:

Task-Referenz-Deklaration ::=
{ **DECLARE | DCL** } Bezeichner-Angabe [Dimensions-Attribut]
[Zuweisungsschutz] **REF TASK** [Global-Attribut] [Initialisierungsattribut] ;

Die allgemeine Form der Spezifikation von Task-Referenz-Variablen lautet:

Task-Referenz-Spezifikation ::=
{ **SPECIFY | SPC** } Bezeichner-Angabe [virtuelle-Dimensionsliste]
[Zuweisungsschutz] **REF TASK** Global-Attribut ;

Das folgende kurze Beispiel zeigt eine Anwendungsmöglichkeit für Task-Referenzvariablen:

Beispiel:

Die beiden Ausschnitte aus zwei PEARL-Programmen zeigen, wie von einer Prozedur aus verschiedene Tasks einmal über deren Namen und einmal über Referenzen gestartet werden können.

```
PROBLEM ;
    SPC  ( Verbraucher_1, Verbraucher_2, Verbraucher_3 )  TASK  GLOBAL ;
    . . .
    starteTask :  PROC  (  Index  FIXED  ) ;
                      CASE  Index
                      ALT          ACTIVATE  Verbraucher_1 ;
                      ALT          ACTIVATE  Verbraucher_2 ;
                      ALT          ACTIVATE  Verbraucher_3 ;
                      FIN ;
          END ; ! starteTask
    . . .

PROBLEM ;
    SPC  ( Verbraucher_1, Verbraucher_2, Verbraucher_3 )  TASK  GLOBAL ;
    DCL  Verbraucher  ( 3 )        INV  REF  TASK  GLOBAL
                                   INIT  ( Verbraucher_1, Verbraucher_2,
    Verbraucher_3 ) ;
    . . .
    starteTask :  PROC  (  Index  FIXED  ) ;
                            IF  Index  <  UPB Verbraucher THEN
                              ACTIVATE  Verbraucher  (  Index  ) ;
                            FIN ;
          END ; ! starteTask
    . . .
```

9.1.2 Bestimmung von Task-Adressen

Die Adresse einer Task erhält man durch den Aufruf der vordefinierten Funktion TASK oder einfach durch Zuweisung eines Task-Bezeichners an eine Task-Referenzvariable.

Beispiel:

```
SPC  Temperatur_Messung  TASK  GLOBAL ;
DCL  ptr_task  REF  TASK ;
...
ptr_task :=  TASK  ( Temperatur_Messung ) ;
ptr_task  := Temperatur_Messung ;
```

Beide Zuweisungen sind äquivalent, jedoch sollte die erste Form genutzt werden, da sie die Adresszuweisung besser dokumentiert.

Die Adresse der laufenden Task erhält man durch Aufruf der Funktion TASK ohne eine Parameterangabe.

Beispiel:

```
...
ptr_task := TASK ;        ! liefert Adresse der Task, in der diese Anweisung steht.
...
```

Neben der Verwendung von Task-Referenzvariablen in Task-Anweisungen können diese auch zur Identifizierung von Tasks benutzt werden (vergl. IS- und ISNT-Operatoren).

Beispiel:

```
IF   ptr_task  IS  TASK (hello)   THEN
        ...
FIN ;
IF   ptr_task  IS  TASK   THEN
        /* zeigt auf aktuelle Task */
        ...
FIN ;
```

9.1.3 Bestimmung von Task-Prioritäten

Mit der vordefinierten Funktion PRIO kann die (aktuelle) Priorität einer Task ermittelt werden. Ohne eine Parameterangabe liefert sie die Priorität der gerade laufenden Task, mit einem Task-Namen als Parameter liefert sie deren Priorität zurück.

Beispiel:

```
DCL  akt_prio  FIXED (15) ;
DCL  prio_task_a  FIXED (15) ;
SPC  task_a  TASK  GLOBAL ;
...
akt_prio := PRIO ;
prio_task_a := PRIO (task_a) ;
CONTINUE  task_a  PRIO  akt_prio + 1 ;
```

9.2 Anweisungen zur Steuerung von Tasks

Eine Task kann gestartet, beendet, angehalten, fortgesetzt, verzögert und ausgeplant werden.

9.2.1 Startbedingung

Tasks werden nicht wie Prozeduren durch Aufruf unmittelbar, sondern in Abhängigkeit von Zeitpunkten und Interrupts ausgeführt. Eine Task muß deshalb zunächst durch eine Task-Steueranweisung der Syntax

```
Task_Starten ::=
        [ Startbedingung ] ACTIVATE Name§Task;
```

zum Start **eingeplant** werden. Der Start selbst wird vom Realzeit-Betriebssystem verwaltet: Ist die in der Steueranweisung angegebene Startbedingung erfüllt, so wird die Task zunächst "lauffähig"; erst wenn sie auch höchstpriore lauffähige Task ist, wird sie gestartet, d.h. in den Zustand "laufend" überführt (vgl. auch 9.2.2).

Beispiele:

AT 20:0:0 **ACTIVATE** Statistik;

bedeutet, daß die Task "Statistik" erst zum zukünftigen Zeitpunkt "20 Uhr" gestartet werden soll.

ACTIVATE Statistik;

bedeutet hingegen, daß die Task "Statistik" unmittelbar gestartet werden soll.

Mit der Startbedingung können gemäß der Syntax-Definition

```
Startbedingung ::=
                  AT    Ausdruck§Uhrzeit  [ Frequenz ]
                | AFTER   Ausdruck§Dauer  [ Frequenz ]
                | WHEN   Name§Interrupt    [ AFTER Ausdruck§Dauer ]  [ Frequenz ]
                | Frequenz

Frequenz ::=
        ALL  Ausdruck§Dauer  [    { UNTIL  Ausdruck§Uhrzeit }
                                | { DURING  Ausdruck§Dauer }  ]
```

Tasks auch zum zyklischen (wiederholten) Start eingeplant werden.

Beispiele:

ALL T **ACTIVATE** Regler;
bedeutet, daß die Task “Regler“ zu jedem ganzen Vielfachen des Zeitintervalls “T“ lauffähig wird.

WHEN Alarm **ACTIVATE** Abschaltung;
bedeutet, daß die Task “Abschaltung“ bei jedem Auftreten des Interrupts “Alarm“ lauffähig wird.

Durch AT Ausdruck§Uhrzeit wird festgelegt, zu welcher Uhrzeit die Task erstmalig ausgeführt werden soll, durch AFTER Ausdruck§Dauer, ab wann dies relativ zur Ausführung der Task-Steueranweisung geschehen soll, und durch WHEN Name§Interrupt, daß dies beim Eintreffen des angegebenen Interrupts, evtl. verzögert um die in AFTER Ausdruck§Dauer genannte Dauer, erfolgen soll. Fehlen AT, AFTER und WHEN, so wird die Task sofort nach Ausführung der Task-Steueranweisung lauffähig.

Soll die Task in gleichen zeitlichen Abständen wiederholt werden, so ist die Zeit zwischen zwei Ausführungen durch ALL Ausdruck§Dauer festzulegen. Um die wiederholten Ausführungen zu begrenzen, kann mit UNTIL Ausdruck§Uhrzeit eine Uhrzeit oder mit DURING Ausdruck§Dauer eine Zeitdauer festgelegt werden, nach der keine Task-Starts mehr wiederholt werden.

Beginnt die Startbedingung mit WHEN, so wird die Task bei jedem Eintreffen des angegebenen Interrupts unter Berücksichtigung der weiteren Komponenten der Startbedingung erneut lauffähig. Eine durch ALL bestimmte Einplanung wird relativ zum Eintreffen des Interrupts neu wirksam, die vorherige unwirksam.

Die Startbedingung wird unwirksam (d.h. die mit ihr verknüpfte Task wird nicht mehr ausgeführt),

- wenn sie mit AFTER oder ALL beginnt und eine durch UNTIL oder DURING gesetzte Endebedingung erreicht ist oder wenn sie die Form AFTER Ausdruck§Dauer hat und die angegebene Dauer (vom Überlaufen der Task-Steueranweisung an gerechnet) verstrichen ist,
- durch Ausführung einer Anweisung zum Ausplanen einer Task (siehe 9.2.7),
- beim Überlaufen einer neuen Task-Steueranweisung desselben Typs. Die alte Startbedingung wird dabei durch die neue ersetzt oder, falls keine neue angegeben ist, gelöscht.

Beispiel:

```
...
AT   12:0:0   ACTIVATE   Protokoll ;
ALL   2 HRS   ACTIVATE   Protokoll ;
...
```

Diese beiden Einplanungen können nicht zur selben Zeit gelten: Die Startbedingung ALL 2 HRS ersetzt vielmehr die Startbedingung AT 12:0:0, sofern nicht vor der Ausführung der zweiten Anweisung der Zeitpunkt “12 Uhr“ eintritt.

9.2.2 Starten einer Task (ACTIVATE)

Die allgemeine Form der Anweisung zum Start einer Task (d.h. zur Einplanung einer Task zum Start) lautet:

```
Task-Starten ::=
        [ Startbedingung ]   ACTIVATE   Name§Task   [ Priorität ] ;

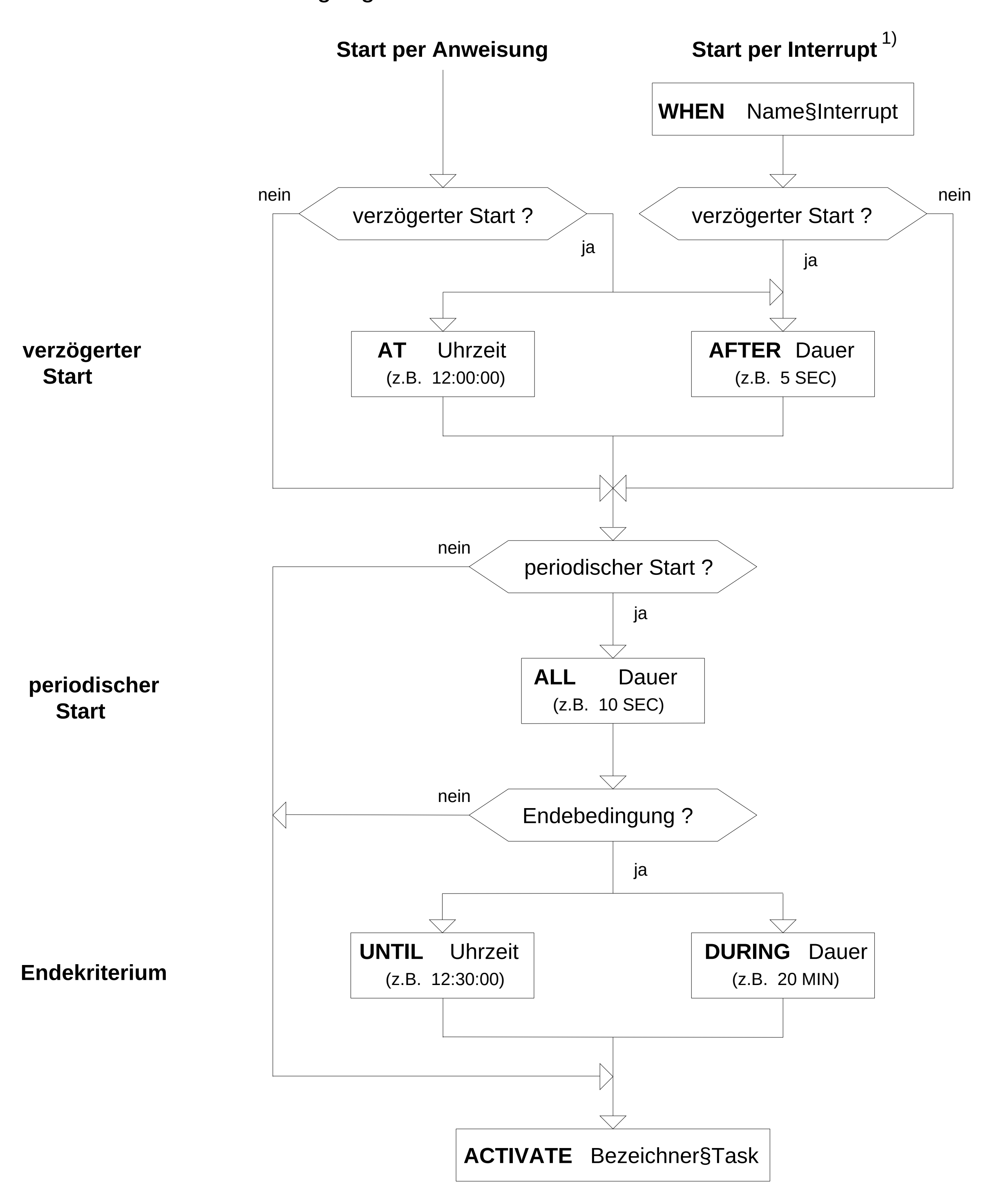
Priorität ::=
        { PRIORITY | PRIO }  Ausdruck§mit-positiver-ganzer-Zahl-als-Wert
```

Als Folge der Ausführung einer solchen Start-Anweisung bewirbt sich die bezeichnete Task sofort (Form ohne Startbedingung) oder zu dem durch die Startbedingung bestimmten Zeitpunkt um die Zuteilung eines Prozessors - konkurrierend mit allen anderen Tasks, die sich zu dem Zeitpunkt des Starts ebenfalls um diesen Prozessor bewerben (lauffähig sind). Beim sofortigen Start konkurriert die gestartete Task also insbesondere mit der startenden Task, wenn nur ein Prozessor zur Verfügung steht.

Möglicherweise ist die bezeichnete Task bei Ausführung der Start-Anweisung bereits gestartet und noch nicht beendet. Ist in diesem Fall keine Startbedingung angegeben, so erfolgt eine Fehlermeldung. Enthält die Start-Anweisung jedoch eine Startbedingung, so wird die bezeichnete Task weitergeführt und ihr erneuter Start entsprechend der Startbedingung eingeplant (gepuffert), wobei eine ggf. bereits bestehende Einplanung durch die neue ersetzt wird.

Eine eventuell angegebene Priorität überschreibt die in der Vereinbarung der bezeichneten Task angegebene Priorität.

Das nachfolgende Bild veranschaulicht die Möglichkeiten, eine Task direkt oder unter verschiedenen Startbedingungen zu starten.



1) Nach dem Eintreffen des Interrupts wird die Task unter der zugehörigen Startbedingung wieder neu eingeplant. Die alte Einplanung wird unwirksam.

Eine Task wird beendet

- wenn sie die abschließende END-Anweisung ihres Körpers erreicht oder
- durch Ausführung einer auf sie bezogenen Anweisung zum Beenden von Tasks (siehe 9.2.3).

Beispiel:

Die Task "Druckmessung" soll alle 5 Sekunden den Druck in einem Behälter messen und an die Task "Kontrolle" übergeben; steigt der Druck verdächtig schnell an, soll die Messung jede Sekunde mit erhöhter Priorität durchgeführt werden. Die Task "Kontrolle" wird durch die Task "Initial" unmittelbar gestartet.

```
PROBLEM :
    Initial :  TASK  MAIN ;
               ACTIVATE   Kontrolle ;
                   /* weitere Initialisierungen */

               END ; ! Initial

    Kontrolle :  TASK   PRIORITY  6 ;
               ALL   5 SEC   ACTIVATE   Druckmessung ;
                   /* Uebernahme der Meßwerte
                   Falls der Druck steigt : */
               ALL   1 SEC   ACTIVATE   Druckmessung   PRIO  2 ;
               ...
               END ; ! Kontrolle

    Druckmessung :  TASK   PRIO  5 ;
                   /* Messung und Uebergabe an Kontrolle */
               END ; ! Druckmessung

    ...
```

9.2.3 Beenden einer Task (TERMINATE)

Das vorzeitige Beenden einer Task wird durch die folgende Anweisung erreicht:

Task-Beenden ::=
    TERMINATE [Name§Task] ;

Fehlt die Angabe Name§Task, so bezieht sich die Anweisung auf die Task, in deren Körper sie enthalten ist.

Der beendeten Task werden alle von ihr belegten Betriebsmittel (einschließlich Prozessor) entzogen. Von der Task gesperrte Synchronisiervariablen werden jedoch nicht automatisch freigegeben.

9.2.4 Anhalten einer Task (SUSPEND)

Durch Ausführung der Anweisung

Task-Anhalten ::=
    SUSPEND [Name§Task] ;

wird die angegebene Task - bzw. die ausführende Task, wenn Name§Task fehlt -, angehalten, d.h. ihre Ausführung wird zurückgestellt. Der ihr zugeordnete Prozessor wird ihr entzogen - nicht aber alle anderen von ihr belegten Betriebsmittel.

Eine angehaltene Task kann nur durch die Ausführung einer Fortsetzungsanweisung fortgesetzt werden (siehe 9.2.5).

9.2.5 Fortsetzen einer Task (CONTINUE)

Eine angehaltene Task kann durch die folgende Anweisung sofort, zu einer bestimmten Uhrzeit, nach einer bestimmten Zeitdauer oder nach Eintritt eines Interrupts fortgesetzt werden:

Task-Fortsetzen ::=
    [einfache-Startbedingung] **CONTINUE** [Name§Task] [Priorität]

einfache-Startbedingung ::=
    AT Ausdruck§Uhrzeit | **AFTER** Ausdruck§Dauer | **WHEN** Name§Interrupt

Ist Name§Task angegeben, so bewirkt die Anweisung, daß sich die bezeichnete Task sofort (Form ohne Startbedingung) oder zu dem durch die Startbedingung bestimmten Zeitpunkt um den ihr zugeordneten Prozessor bewirbt - gegebenenfalls mit der angegebenen Priorität, die dann die vereinbarte Priorität ersetzt.

Die Form ohne Name§Task bewirkt, daß sich die ausführende Task zu dem durch die Startbedingung bestimmten Zeitpunkt erneut um den Prozessor bewirbt - gegebenenfalls mit der angegebenen Priorität, die dann die vereinbarte Priorität ersetzt.

Beispiel:

> Die Task "Erfassung" soll eine Ausgabe veranlassen und sich erst nach Eintritt des Interrupts "Weiter" fortsetzen, dann jedoch mit erhöhter Priorität.

```
Erfassung:    TASK   PRIO 8 ;
                            ! Erfassung von Daten
                     WHEN  Weiter  CONTINUE  PRIO 5 ;
                            ! Ausgabe
                     SUSPEND ;
                            ! Erfassung von Daten
              END ; ! Erfassung
```

In Bild 9-1 sind die durch Ausführung von CONTINUE-Anweisung möglichen Zustands-Übergänge einer Task graphisch veranschaulicht. Für den Fall von "Vormerkungen" sind zusätzlich die Zustandsübergänge bei SUSPEND-Anweisungen angegeben.

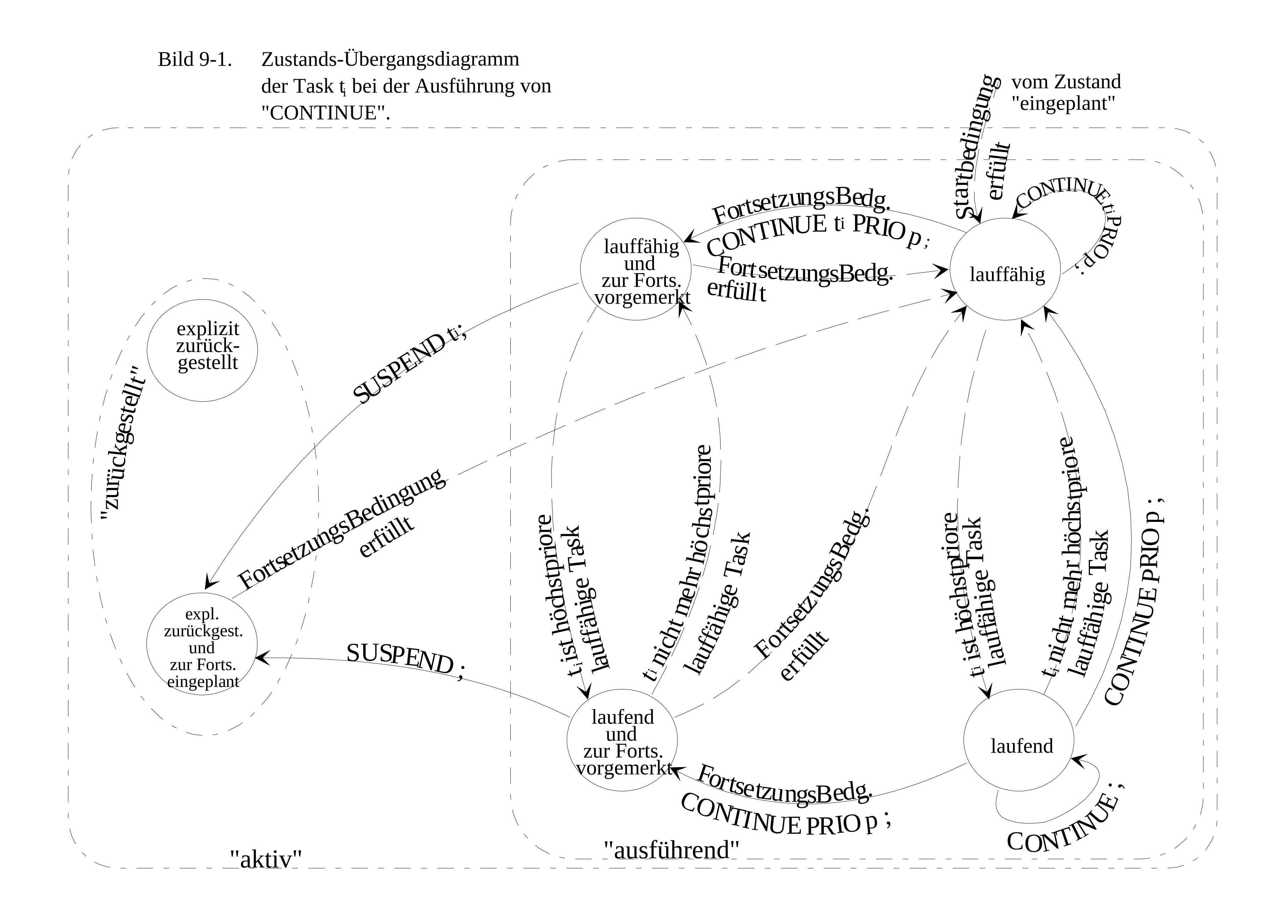


Bild 9-1. Zustands-Übergangsdiagramm der Task t_i bei der Ausführung von "CONTINUE".

9.2.6 Verzögern einer Task (RESUME)

Soll die aktuelle Task für eine bestimmte Zeitdauer oder bis zum Eintritt einer bestimmten Uhrzeit bzw. eines Interrupts den ihr zugeordneten Prozessor freigeben (sich verzögern), so ist folgende Anweisung auszuführen:

Task-Verzögern ::=
  einfache-Startbedingung **RESUME** ;

Diese Anweisung ist äquivalent der ununterbrechbaren Kombination der Anweisungen

einfache-Startbedingung **CONTINUE** ;
SUSPEND ;

Nach Ausführung der Anweisung Task-Verzögern ist die Startbedingung unwirksam, d.h. die Ausführung ist einmalig.

Beispiel:

Die Task “Steuerung“ soll ein Gerät einschalten und 10 Sekunden später prüfen, ob das Gerät wie vorgesehen arbeitet.

```
Steuerung:    TASK ;
                        ! Einschalten des Geräts
                AFTER  10 sec  RESUME ;
                        ! Prüfung der Funktion des Geräts
                ...
              END ; ! Steuerung
```

9.2.7 Ausplanen einer Task (PREVENT)

Mitunter ist es nötig, die für eine Task bestehenden Einplanungen aufzuheben, d.h. dafür zu sorgen, daß die Startbedingungen unwirksam werden, die mit dieser Task verknüpft sind. Dies kann mit folgender Anweisung erreicht werden:

Task-Ausplanen ::=
  PREVENT [Name§Task] ;

Die Anweisung beendet die betreffende Task nicht; ist Name§Task nicht angegeben, so wirkt sie auf die laufende Task.

Beispiel:

Die Prozedur “Steuerung“, die von einer übergeordneten Task aus aufgerufen wird, gibt einen Fahrbefehl an ein Förderzeug aus, das innerhalb von 30 sec die Fertigmeldung “Fertig“ auslösen muß, die den Start der Task “Versorgung“ auslösen soll. Ist nach 30 sec die Fertigmeldung nicht eingetroffen, soll die Task “Stoerung“ gestartet und der eventuell noch mögliche, aber verspätete Start von “Versorgung“ verhindert werden. Im Normalfall - “Fertig“ trifft innerhalb von 30 sec ein - muß der eingeplante Start von “Stoerung“ wieder ausgeplant werden.

```
PROBLEM:
    SPECIFY  Fertig  INTERRUPT ;

    Steuerung: PROC  (     X  FIXED,      /* X-Koordinate */
                           Y  FIXED ) ;   /* Y-Koordinate */
                   /* Umwandlung der übernommenen Koordinaten in einen Bitstring,
                   Ausgabe des Bitstrings an das Förderzeug */
                   WHEN  Fertig  ACTIVATE  Versorgung ;
                   AFTER  30 SEC  ACTIVATE  Stoerung ;
                   ...
           END ; ! Steuerung

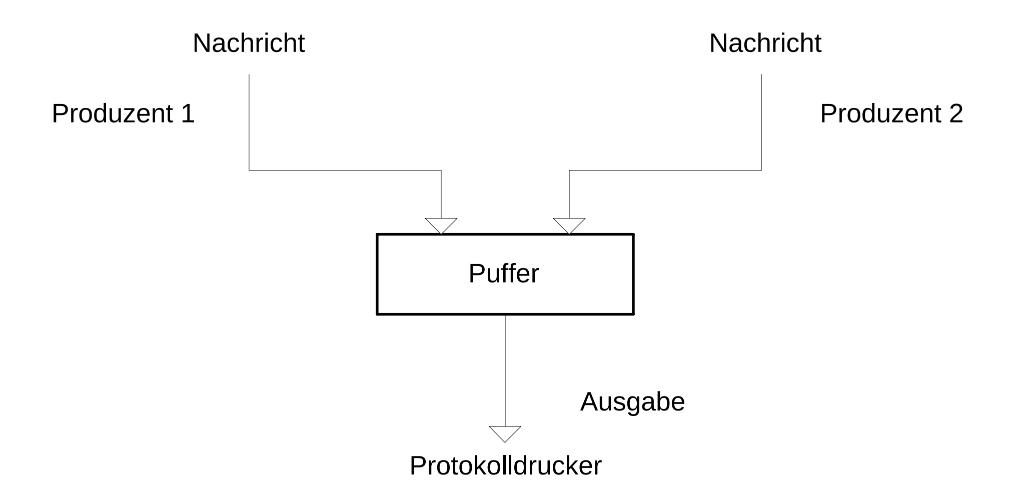
    Versorgung:  TASK  PRIO 3 ;
                  PREVENT  Stoerung ;
                  ...
           END ; ! Versorgung

    Stoerung:  TASK  PRIO 2 ;
                  PREVENT  Versorgung ;
                  ...
           END ; ! Stoerung
```

9.3 Synchronisierung von Tasks

Grundsätzlich werden Tasks unabhängig voneinander abgearbeitet. Es kann jedoch auch der Fall eintreten, daß mehrere Tasks Teilaufgaben eines komplexeren Gesamtproblems bearbeiten und dabei bestimmte Betriebsmittel, insbesondere Daten, gemeinsam benutzen müssen.

Beispiel:



Die Tasks Produzent 1 und Produzent 2 produzieren Nachrichten, die sie in einem gemeinsamen Puffer ablegen wollen; durch die Task Ausgabe wird eine Nachricht aus dem Puffer entnommen und auf einem Protokolldrucker formatiert ausgegeben.

Damit die Operationen richtig ablaufen, müssen sie wie folgt koordiniert werden:

- Ausgabe erzeugt nur dann eine Ausgabe, wenn eine Nachricht gepuffert wurde, wartet also gegebenenfalls auf Füllung des Puffers durch Produzent 1 oder Produzent 2.

- Produzent 1 und Produzent 2 können nur dann eine Nachricht puffern, wenn gerade keine frühere Nachricht durch Ausgabe ausgegeben wird, d.h. sie müssen eventuell zurückgestellt werden, bis Ausgabe eine Nachricht vollständig ausgegeben hat.

- Die Teilabläufe von Produzent 1 und Produzent 2 zum Puffern einer Nachricht schließen sich wechselseitig aus.

Zur Kontrolle solcher Koordinierungsarbeiten stehen zwei Typen von Synchronisiervariablen, die Sema- und die Bolt-Variablen, zur Verfügung. Soll der Zugriff auf gemeinsam benutzte Betriebmittel (z.B. Daten, Prozeduren, Geräte) koordiniert werden, ordnet man diesen Betriebsmitteln jeweils eine Synchronisiervariable zu; die benutzenden Tasks führen sodann vor Benutzung eines Betriebsmittels eine Sperr-Anweisung und nach Benutzung eine Freigabe-Anweisung auf die zugeordnete Synchronisiervariable aus.

Die Zuordnung von Synchronisiervariablen zu Betriebsmitteln erfolgt nicht durch PEARL-Anweisungen; sie ist gedacht und besteht nur darin, daß eine bestimmte Synchronisiervariable immer nur im Zusammenhang mit einem bestimmten Betriebsmittel benutzt wird. Eine solche Zuordnung ist nicht auf Daten, Prozeduren, Geräte etc. beschränkt, sie muß mitunter auch für Programmabschnitte (z.B. zur Erledigung von Teilaufgaben) vorgenommen werden, deren Abläufe alle beendet sein sollen, bevor ein bestimmter Programmabschnitt (z.B. die Hauptaufgabe) weiter ausgeführt wird.

9.3.1 Semaphor-Variablen (SEMA) und -Anweisungen (REQUEST, RELEASE, TRY)

Sema-Variablen können als Werte nichtnegative ganze Zahlen besitzen. Diese Zahlen stellen die Zustände "frei" oder "gesperrt" dar: Null bedeutet den Zustand "gesperrt" (die Sema-Variable "sperrt"), positive Zahlen bedeuten den Zustand "frei". Diese Zustände können nur durch spezielle Sperr- und Freigabe-Anweisungen geändert werden.

Sema-Variablen werden wie folgt deklariert:

DCL Bezeichner-Angabe **SEMA** ;

Beispiel:

DCL (Ein, Aus) **SEMA** ;

Nach ihrer Deklaration hat eine Sema-Variable den Zustand "gesperrt".

Soll eine Sema-Variable explizit den Zustand "gesperrt" erhalten, ist folgende Sperr-Anweisung auszuführen:

REQUEST Name§Sema ;

Die Wirkung dieser Anweisung hängt von dem aktuellen Wert der bezeichneten Sema-Variablen ab:

- Ist der Wert größer Null, wird er um 1 erniedrigt.
- Ist der Wert gleich Null, bleibt er unverändert; die ausführende Task wird jedoch angehalten und in eine Warteschlange eingereiht, die der Sema-Variablen (intern) zugeordnet ist (die Task wird blockiert).

Die Ausführung der Freigabe-Anweisung

RELEASE Name§Sema ;

bewirkt, daß der Wert der bezeichneten Sema-Variablen um 1 erhöht wird. Außerdem werden die in der Warteschlange der Sema-Variablen stehenden Tasks befreit; diese wiederholen ihre Sperr-Anweisung in der Reihenfolge ihrer Priorität.

Beispiel:

Mit diesen Hilfsmitteln läßt sich nun das Problem der Pufferung von Nachrichten folgendermaßen lösen:

Für die Eingabe von Nachrichten in den Puffer wird eine Sema-Variable In_den_Puffer, für die Ausgabe aus dem Puffer eine Sema-Variable Aus_dem_Puffer vereinbart. Bei der Vereinbarung werden In_den_Puffer und Aus_dem_Puffer implizit mit "gesperrt" initialisiert. Die Task Ausgabe wird von außen gestartet. Bevor sie die Tasks Produzent_1, Produzent_2 startet, muß In_den_Puffer den Zustand "frei" erhalten.

```
PROBLEM ;
    DCL  (In_den_Puffer, Aus_dem_Puffer)  SEMA ;

    Ausgabe:        TASK ;
                        RELEASE  In_den_Puffer;
                        ACTIVATE  Produzent_1;
                        ACTIVATE  Produzent_2;
                        REPEAT
                            REQUEST  Aus_dem_Puffer;
                                ! Ausgabe auf  den Protokolldrucker
                            RELEASE  In_den_Puffer ;
                        END ; ! Schleife
                    END ; ! Ausgabe
```

```
Produzent_1: TASK ;
             REPEAT
                      ! Aufbereiten der Nachricht
                  REQUEST  In_den_Puffer;
                      ! Puffern
                  RELEASE  Aus_dem_Puffer ;
             END ; ! Schleife
          END ; ! Produzent_1

Produzent_2: TASK ;
             REPEAT
                      ! Aufbereiten der Nachricht
                  REQUEST  In_den_Puffer;
                      ! Puffern
                  RELEASE  Aus_dem_Puffer ;
             END ; ! Schleife
          END ; ! Produzent_2
```

Erklärung:
Ausgabe wartet zunächst auf die Freigabe von Aus_dem_Puffer, d.h. auf die Pufferung einer Nachricht, weil Aus_dem_Puffer nur von Produzent_1 und Produzent_2 freigegeben werden kann. Nach erfolgter Ausgabe gibt Ausgabe In_den_Puffer frei, läßt also eine Pufferung zu, und wartet auf einen neuen Auftrag. Produzent_1 und Produzent_2 können nur dann eine Nachricht puffern, wenn für In_den_Puffer der Zustand "frei" definiert ist; nach dem Puffern einer Nachricht geben sie Aus_dem_Puffer frei und stoßen damit Ausgabe an.

Der Zustand einer Sema-Variablen kann mit dem monadischen Operator

TRY Name§Sema

erfragt werden. Der Operator liefert als Ergebnis einen BIT(1)-Wert, der in allen logischen Ausdrücken verwendet werden kann. Wenn die Sema-Variable "frei" ist, führt der Operator eine REQUEST-Anweisung aus und liefert den Wert '1'B als Ergebnis. Falls die Sema-Variable bereits den Zustand "gesperrt" hat, wird keine REQUEST-Anweisung ausgeführt und der Wert '0'B geliefert. Die laufende Task wird bei der Ausführung des TRY-Operators nicht blockiert!

Beispiel:

In einem Dialogsystem darf die Funktion “function_1“ nur einmal aktiv sein, die Task “Dialog“ soll aber nicht blockiert werden.

```
PROBLEM ;
    DCL   func_1   SEMA ;

    Dialog :   TASK ;
            . . .
            IF   TRY  func_1   THEN
                ! Sema-Variable  func_1  hatte den Zustand "frei"
                CALL   function_1 ;              ! Funktion ausfuehren
                RELEASE   func_1 ;               ! Sema wieder freigeben
            ELSE
                ! Sema-Variable  func_1  hatte den Zustand "gesperrt"
                PUT  'Funktion 1 zur Zeit nicht moeglich!'  TO  Terminal ;
            FIN ;
            . . .
        END ; ! Dialog
```

Durch fehlerhafte Synchronisierungen können sich Tasks gegenseitig ständig blockieren; eine solche Systemverklemmung (engl. deadlock) könnte beispielsweise durch folgende Programmorganisation entstehen:

Task T1	Task T2
REQUEST S1 ;	**REQUEST** S2 ;
1. Abschnitt	1. Abschnitt
REQUEST S2 ;	**REQUEST** S1 ;
2. Abschnitt	2. Abschnitt
RELEASE S2 ;	**RELEASE** S1 ;
RELEASE S1 ;	**RELEASE** S2 ;

Befinden sich die Tasks T1 und T2 zugleich in ihrem ersten Abschnitt, so entsteht eine Verklemmung, denn einerseits wird T1 durch die Ausführung der Anweisung "REQUEST S2;" blockiert, weil T2 bereits die Sperre mit der Sema-Variablen S2 aufgebaut hat, und andererseits wird T2 durch Ausführung der Anweisung "REQUEST S1 ;" blockiert. Falls keine andere Task eine Freigabe ausführt, bleiben die Blockierungen ständig bestehen.

Zur Vermeidung solcher Situationen können bei Sperr-Anweisungen Listen von Sema-Variablen angegeben werden.

Durch Ausführung der Anweisung

REQUEST Name§Sema [, Name§Sema] ··· ;

wird die laufende Task angehalten, falls für eine der bezeichneten Sema-Variablen der Zustand "gesperrt" notiert ist; die Task wird erst wieder fortgesetzt, wenn keine der Sema-Variablen mehr sperrt. Sperrt keine der bezeichneten Sema-Variablen, so wird die ausführende Task fortgesetzt, nachdem die Werte dieser Sema-Variablen um 1 erniedrigt wurden.

Die Anweisung

RELEASE Name§Sema [, Name§Sema] ··· ;

wirkt, als ob für jede der bezeichneten Sema-Variablen in ungeteilter Form eine Freigabe-Anweisung ausgeführt würde.

Durch folgende Programmorganisation ließe sich die Verklemmung im letzten Beispiel vermeiden:

Task 1	Task 2
REQUEST S1, S2 ;	**REQUEST** S1, S2 ;
1. Abschnitt	1. Abschnitt
2. Abschnitt	2. Abschnitt
RELEASE S1, S2 ;	**RELEASE** S1, S2 ;

Die Reihenfolge der Sema-Variablen in den Listen ist ohne Bedeutung.

Nach dem TRY-Operator darf nur eine einzelne Sema-Variable angegeben werden, die Angabe einer Liste von Sema-Variablen ist nicht möglich.

Die allgemeine Formen für die Deklarationen von Sema-Variablen sowie Sperr- und Freigabe-Anweisungen lauten:

Sema-Deklaration ::=
{ **DECLARE | DCL** } Bezeichner-Angabe [Dimensionsattribut] **SEMA** [Global-Attribut]
[**PRESET** (ganze-Zahl-ohne-Genauigkeit [, ganze-Zahl-ohne-Genauigkeit] ···)] ;

Request-Anweisung ::=
REQUEST Name§Sema [, Name§Sema] ··· ;

Release-Anweisung ::=
RELEASE Name§Sema [, Name§Sema] ··· ;

TRY-Operator ::=
TRY Name§Sema

Es sind also auch Felder von Sema-Variablen möglich; die einzelnen Elemente werden in den Sperr- und Freigabe-Anweisungen indiziert angesprochen.

Sema-Variablen müssen auf Modulebene vereinbart werden. Bei der Deklaration können Sema-Variablen auch explizit durch PRESET-Angabe initialisiert werden; die betreffenden Sema-Variablen erhalten die angegebenen Werte entsprechend der Reihenfolge zugewiesen.

Beispiel:

Den Sema-Variablen S1 und S2 sollen bei ihrer Deklaration die Werte 3 und 5 zugewiesen werden.

DCL (S1, S2) **SEMA** **PRESET** (3, 5) ;

Die allgemeine Form für die Spezifikation von Sema-Variablen lautet:

Sema-Spezifikation ::=
{ **SPECIFY** | **SPC** } Bezeichner-Angabe
[virtuelles-Dimensionsattribut] **SEMA**
{ Global-Attribut | Identifikationsattribut } ;

9.3.2 Bolt-Variablen (BOLT) und -Anweisungen (ENTER, LEAVE, RESERVE, FREE)

Angenommen, in verschiedenen Tasks werden dieselben Daten zur Rechnung benutzt, aber nicht modifiziert (z.B. Vergleichsgrößen für Überwachungsprozesse); zusätzlich soll eine andere Task diese Daten neu ermitteln (z.B. Vorgabe neuer Überwachungsdaten). Es soll gewährleistet sein, daß sich die Abläufe zur Modifikation und zur Benutzung der Daten gegenseitig ausschließen; dagegen ist simultane Benutzung der Daten (durch mehrere Tasks) erwünscht.

Diese Probleme lassen sich im Prinzip mit den bisher beschriebenen Sperr- und Freigabe-Anweisungen lösen; jedoch ist die Formulierung relativ kompliziert und die Laufzeit für die Ausführung erheblich. Deshalb werden weitere vier Anweisungen geboten, die auf Variablen vom Datentyp BOLT arbeiten.

Eine Boltvariable kann die Zustände "gesperrt", "Sperre möglich" oder "Sperre nicht möglich" annehmen, je nachdem, ob das zugeordnete Betriebsmittel exklusiv benutzt wird, frei ist oder simultan benutzt wird.

Bolt-Variablen werden z.B. so deklariert:

DCL Bezeichner-Angabe **BOLT** ;

Nach ihrer Deklaration hat eine Bolt-Variable den Zustand "Sperre möglich".

Einem Betriebsmittel sei die Bolt-Variable B zugeordnet. Bei Eintritt in einen kritischen Abschnitt zur exklusiven Benutzung dieses Betriebsmittels wird der Zugriff anderer Tasks durch Ausführung der Anweisung

RESERVE B ;

gesperrt. Die Freigabe bei Austritt aus diesem kritischen Abschnitt erfolgt durch die Anweisung

FREE B ;

Die kritischen Abschnitte zur simultanen Benutzung des Betriebsmittels werden durch Ausführung der Anweisung

ENTER B ; bzw. **LEAVE** B ;

eingeleitet bzw. abgeschlossen.

Die genauere Beschreibung der Wirkung dieser Anweisungen berücksichtigt den Zustand und die Modifikation der Bolt-Variablen:

Wirkung von RESERVE B;

Hat B den Zustand "Sperre möglich", erhält B den Zustand "gesperrt"; anderenfalls wird die ausführende Task angehalten und in eine Warteschlange eingereiht, die B zugeordnet ist.

Wirkung von FREE B;

B bekommt den Zustand "Sperre möglich". Außerdem werden alle aufgrund einer RESERVE-Anweisung in der Warteschlange von B wartenden Tasks befreit; diese wiederholen ihre Sperr-Anweisungen in der Reihenfolge ihrer Priorität. Wartet keine Task aufgrund einer RESERVE-Anweisung, so werden alle anderen wartenden Tasks befreit, die dann ihrerseits die ENTER-Anweisung in der Reihenfolge ihrer Priorität wiederholen.

Wirkung von ENTER B;

Hat B den Zustand "gesperrt", oder befindet sich in der Warteschlange von B eine Task aufgrund einer RESERVE-Anweisung, so wird die ausführende Task angehalten und in die Warteschlange von B eingereiht. Anderenfalls erhält B den Zustand "Sperre nicht möglich", um exklusiven Zugriff zu verbieten; außerdem wird die (intern notierte) Anzahl Z der “benutzenden" Tasks um 1 erhöht.

(D.h. Tasks, die "ENTER B" ausführen, werden erst nach den Tasks fortgesetzt, die einen exklusiven Zugriff auf das Betriebsmittel benötigen.)

Wirkung von LEAVE B;

Ist Z = 1, so wirkt diese Anweisung wie "FREE B;". Andernfalls wird Z um 1 erniedrigt, und B behält den Zustand "Sperre nicht möglich".

Beispiel:

Eine Task “Messung“ ermittelt laufend Werte von Vergleichsgrößen aus einem zu überwachenden Prozeß, die von den Tasks “Steuerung“ und “Disposition“ für Berechnungen benötigt werden. Es soll gewährleistet sein, daß “Messung“ nur dann die Vergleichsgrößen ändert, wenn sie nicht benutzt werden; dagegen sollen “Steuerung“ und “Disposition“ die Vergleichsgrößen simultan benutzen.

Hierzu wird eine Bolt-Variable Vwert deklariert; in den Körpern der drei Tasks werden die kritischen Abschnitte der Modifikation oder Benutzung der Vergleichsgrößen folgendermaßen eingeleitet bzw. abgeschlossen:

Im Körper von Messung:

```
...
RESERVE   Vwert ;
          ! Modifikation
FREE   Vwert ;
...
```

In den Körpern von Steuerung und Disposition:

```
...
ENTER   Vwert ;
          ! Benutzung
LEAVE   Vwert ;
...
```

Alle Anweisungen für Bolt-Variable sind auch für Listen von Bolt-Variablen definiert - in Analogie zu den Sperr- und Freigabe-Anweisungen für Sema-Variablen.

Allgemein können Bolt-Variablen wie folgt vereinbart und benutzt werden:

Bolt-Deklaration ::=
{ **DECLARE | DCL** } Bezeichner-Angabe [Dimensionsattribut] **BOLT**
[Global-Attribut] ;

Bolt-Spezifikation ::=
{ **SPECIFY | SPC** } Bezeichner-Angabe [virtuelles-Dimensionsattribut]
BOLT { Global-Attribut | Identifikationsattribut } ;

Bolt-Anweisung ::=
RESERVE Name§Bolt [, Name§Bolt] ··· ;
| **FREE** Name§Bolt [, Name§Bolt] ··· ;
| **ENTER** Name§Bolt [, Name§Bolt] ··· ;
| **LEAVE** Name§Bolt [, Name§Bolt] ··· ;

Es können also auch Felder von Bolt-Variablen vereinbart werden.
Bolt-Vereinbarungen müssen auf Modulebene erfolgen.

9.4 Interrupts und Interrupt-Anweisungen

9.4.1 Vereinbarung von Interrupts und Software-Interrupts

Ein Interrupt (Unterbrechung) ist eine Meldung des gesteuerten Prozesses über einen (Unterbrechungs-) Eingang des Interruptwerks an das Betriebssystem, das nach Eintritt des Interrupts die vom Programmierer dafür vorgesehene Reaktion einzuleiten hat, z.B.: "Bei Eintritt des Interrupts Fertig soll die Task Nachschub gestartet werden".

Die zur Verfügung stehenden Unterbrechungseingänge eines Rechensystems sind im Benutzerhandbuch unter Angabe ihrer (System-) Namen beschrieben. Die hiervon für ein PEARL-Programm benötigten Unterbrechungseingänge werden im Systemteil deklariert, wobei ihnen Benutzernamen zugeordnet werden können. Unter diesen Benutzernamen werden sie dann als Interrupts im Problemteil spezifiziert, um sie in den Task-Steueranweisungen (siehe 9.2) und den Interrupt-Anweisungen benutzen zu können.

Beispiel:

```
MODULE ;
SYSTEM ;
        Fertig : Hard_Int (7) ; ! Hard_Int ist der Systemname

PROBLEM ;
SPECIFY  Fertig  INTERRUPT ;
...
Initialisierung :  TASK ;
                   ...
                   WHEN  Fertig  ACTIVATE  Nachschub ;
                   ...
                   END ; ! Initialisierung

...
Nachschub :        TASK  PRIORITY 2 ;
                   ! Taskkörper
                   END ; ! Nachschub
...
```

Die allgemeine Form einer Interrupt-Spezifikation lautet:

```
Interrupt-Spezifikation ::=
        { SPECIFY | SPC }  Bezeichner-Angabe  { INTERRUPT | IRPT }
        [Global-Attribut ] ;
```

Die meisten modernen Computer und Betriebssysteme bieten jedoch gar nicht mehr die Möglichkeit Unterbrechungseingänge anzusprechen, da die Aufgaben der hardwarenahen Geräte-Steuerung und -Überwachung von speicherprogrammierbaren Steuerungen übernommen werden. Die von PEARL für Interrupts gebotenen Möglichkeiten sind jedoch auch für sogenannte *Software-Interrupts* nutzbar, die über einen eigenen Namen im Systemteil eingeführt werden. Das Benutzerhandbuch gibt Auskunft über den Systemnamen und Nutzungsmöglichkeiten.

9.4.2 Interrupt-Anweisungen (TRIGGER, ENABLE, DISABLE)

Interrupts sind durch Ihre Vereinbarung zunächst “gesperrt“ und müssen durch eine Enable-Anweisung freigegeben werden.

Will man die Wirkung eines freigegebenen Interrupts, beispielsweise bei Störungen im technischen Prozeß, unterdrücken, d.h. sich auf ihn beziehende Startbedingungen für Task-Anweisungen sollen bei Eintritt des Interrupts unwirksam bleiben, so ist zu diesem Zweck eine Disable-Anweisung auszuführen, die ihrerseits durch Ausführung einer Enable-Anweisung wieder aufgehoben werden kann.

```
Disable-Anweisung ::=
        DISABLE   Name§Interrupt ;

Enable-Anweisung ::=
        ENABLE   Name§Interrupt ;
```

Der Gültigkeitsbereich der Disable-Anweisung beginnt beim Überlaufen der Anweisung und endet mit der Ausführung einer Enable-Anweisung.

Beispiel:

```
MODULE ;
SYSTEM ;
        Alarm : INT ;
        ...
PROBLEM ;

        SPC  Alarm  INTERRUPT ;
        ...
        Init :  TASK  MAIN ;
                WHEN  Alarm  ACTIVATE  Stoerdienst ;
                ...
                END ; ! Init
```

```
        Stoerdienst :  TASK  PRIO  1 ;
                DISABLE  Alarm ;
                        ! Untersuchung der Ursache
                        ! Reaktion einleiten
                ENABLE  Alarm ;
                END ; ! Stoerdienst
        ...
MODEND ;
```

Der Test von Echtzeit-Programmen erfordert mitunter, die Wirkung von Interrupts zu simulieren, zumal wenn der technische Prozeß noch nicht an den Rechner angeschlossen ist. Für solche Simulationen steht die Trigger-Anweisung zur Verfügung:

```
Trigger-Anweisung :=
        TRIGGER  Name§Interrupt ;
```

Beispiel:

```
MODULE ;
SYSTEM ;
     Fertig : Soft_Int ;
     . . .
PROBLEM ;
     . . .
     WHEN  Fertig
          ACTIVATE  Steuerung ;
     . . .
MODEND ;
```

```
MODULE  ( Test ) ;
PROBLEM ;
     SPC  Fertig  IRPT GLOBAL ;

     . . .

     /* z.B. zu zufälligen Zeiten */
     TRIGGER  Fertig ;
      . . .

MODEND ;
```

10. Eingabe und Ausgabe

Die Ein- und Ausgabe-Anweisungen ermöglichen die Übertragung von Daten aus dem Arbeitsspeicher des Rechners auf eine externe Datenstation (Ausgabe) sowie umgekehrt die Übertragung von Daten, die sich auf einer externen Datenstation befinden, in den Arbeitsspeicher (Eingabe). Datenstationen sind primär Geräte der Standardperipherie (Drucker, Konsole, Platte, Magnetband, Tastatur usw.) oder der Prozeßperipherie (Meßwertgeber, Stellglieder, unterlagerte Steuerung usw.).

Auf diesen *system-definierten* Datenstationen können im Programm *benutzerdefinierte* Datenstationen kreiert werden zur Aufnahme von Daten z.B. auf Platten, Magnetbändern, Druckern etc.

In den E/A-Anweisungen des Problemteils werden die Datenstationen unter frei wählbaren, logischen Benutzernamen angesprochen; die Eigenschaften der Datenstationen sind zuvor unter Aufführung dieser Namen zu vereinbaren.

In herkömmlichen Programmiersprachen wird die Zuordnung der im Programm verwendeten logischen Bezeichnungen (Benutzernamen) für Datenstationen zu den Geräten eines speziellen Rechensystems durch zusätzliche Steuerangaben beim Einrichten eines Übersetzer-Programms (Job-Control-Angaben) getroffen. In PEARL erfolgen diese Zuordnungen in einheitlicher Schreibweise durch Vereinbarungen im Systemteil.

10.1 Systemteil

Der Systemteil dient der Beschreibung der verwendeten E/A-Konfigurationen eines PEARL-Programms. Allen Geräten eines Rechnersystems, die direkt mit PEARL E/A-Anweisungen angesprochen werden können, muß ein Systemname zugeordnet sein. Eine Liste dieser Geräte und ihrer Systemnamen findet man im PEARL-Benutzerhandbuch des jeweiligen Zielsystems. Im Systemteil müssen den Systemnamen der benötigten Geräte frei wählbare Benutzernamen zugeordnet werden. Nur über diese Benutzernamen können die Geräte dann in E/A-Anweisungen des Problemteils angesprochen werden. Gibt es mehrere Geräte des gleichen Typs, können diese auch durch einen dem Systemnamen angehängten Index unterschieden werden. Ein Systemname hat also folgende allgemeine Form:

```
System-Name ::=
        Bezeichner [ ( nngz§index ) ]

nngz ::=
        ganze-Zahl-ohne-Genauigkeit§nicht-negativ
```

Für die meisten Prozeßgeräte sind weitere Angaben erforderlich wie z.B. die Kanalnummer, und bei digitalen E/A-Kanälen kann sogar die Auswahl einzelner Bits gewünscht sein. Nach dem Systemnamen ist deshalb für Prozeßgeräte folgende Erweiterung möglich:

```
Erweiterung-Prozess-Gerät ::=
        * nngz§Kanalnummer [ * nngz§Position [ , nngz§Breite ] ]
```

Mit “Position“ kann z.B. die Anfangsbit-Position des Geräteanschlusses einer digitalen Ein-(Aus-)gabe, mit “Breite“ die Anzahl der Bits dieses Geräteanschlusses angegeben werden.

Ein Systemteil besteht also aus einer Anzahl von Gerätezuordnungen folgender Form:

```
Geräte-Zuordnung ::=
        Benutzer-Name : System-Name [ Erweiterung-Prozess-Gerät ]

Benutzer-Name ::=
        Bezeichner
```

Für das unten angegebene Beispiel eines Systemteils wird ein fiktives Rechner-system mit folgenden Systemnamen vorausgesetzt:

System-Name	Gerät
STDIN	Standard Eingabe (Konsole)
STDOUT	Standard Ausgabe (Konsole)
SERIAL	Für E/A mit seriellen Schnittstellen
DISC	Zum Lesen und Schreiben von Dateien (Platte, Floppy, ...)
DIGEA	Adapter für digitale E/A

PEARL-Beispiel für Systemteil:

```
MODULE   (Beispiel) ;
SYSTEM ;
        termin   :   STDIN ;
        termout  :   STDOUT ;
        file     :   DISC ;
        tty_1    :   SERIAL (1) ;
        tty_4    :   SERIAL (4) ;
        counter  :   DIGEA * 0 ;
        schalter:    DIGEA * 1 * 16,1 ;
        motor    :   DIGEA * 1 * 1,4 ;
        ...
PROBLEM  ;
        ...
```

Die verfügbaren Geräte und die zugeordneten Systemnamen müssen dem jeweiligen Benutzerhandbuch einer PEARL-Implementierung entnommen werden. Dort ist auch beschrieben, ob (und wenn ja wie) Systemnamen geändert werden können. Für die meisten PEARL-Systeme gibt es zusätzlich die Möglichkeit, vom Anwender erstellte Gerätetreiber einschließlich ihrer Systemnamen in das PEARL-System zu integrieren und mit PEARL E/A-Anweisungen anzusprechen.

10.2 Vereinbarung von Datenstationen (DATION) im Problemteil

10.2.1 System-Datenstationen

Vor ihrer Benutzung müssen System-definierte Datenstationen unter Angabe ihres zugeordneten Benutzernamens im Problemteil spezifiziert werden. Für die Vereinbarungen aus dem obigen Beispiel kann dies so erfolgen:

```
SPC   termin    DATION  IN      ALPHIC ,
      termout   DATION  OUT     ALPHIC ,
      file      DATION  INOUT   ALL ,
      tty_1     DATION  IN      ALPHIC ,
      tty_4     DATION  INOUT   ALL ,
      counter   DATION  IN      BASIC ,
      schalter  DATION  OUT     BASIC ,
      motor     DATION  OUT     BASIC ;
```

Die allgemeine Form lautet:

Dation-Spezifikation ::=
{ **SPECIFY | SPC** } Bezeichner-Angabe Typ-Dation [Global-Attribut] ;

Die verschiedenen Attribute in einer System-Dation-Spezifikation beschreiben die grundlegenden Eigenschaften des physikalischen Gerätes, mit dem kommuniziert werden soll. Die Eigenschaften eines Gerätes und die möglichen Attribute sind für alle Geräte im Benutzerhandbuch beschrieben.

10.2.2 Benutzerdefinierte Datenstationen

Durch eine Dation-Deklaration wird eine logische, sogenannte benutzerdefinierte Datenstation (oder Benutzer-Dation) auf einem physikalischen Gerät (System-Dation) erzeugt. Die Zuordnung zu einem Gerät wird durch die CREATED-Angabe festgelegt.

Alle in den folgende Kapiteln beschriebenen Ein- und Ausgabeanweisungen beziehen sich auf Benutzer-Dations - die direkte Angabe von System-Dations führt zu Laufzeitfehlern.

Dation-Deklaration ::=
{ **DECLARE | DCL** } Bezeichner-Angabe Typ-Dation [Global-Attribut]
CREATED (Name§system-def-Dation) ;

Es dürfen demnach keine Felder von Datenstationen vereinbart werden. Es ist jedoch möglich, Felder von Referenzen auf Datenstationen zu deklarieren, um Datenstationen auch indiziert ansprechen zu können.

Die verschiedenen Attribute ermöglichen es, bereits zur Übersetzungszeit Widersprüche zwischen den Eigenschaften von Datenstationen und ihrer Benutzungsweise in E/A-Anweisungen festzustellen.

Typ-Dation ::=
DATION Quelle-Senke-Attribut Klassenattribut
[Gliederung] [Zugriffsattribut] [Kontroll-Attribut]

Jede Datenstation ist Quelle und/oder Senke einer Datenübertragung. Die betreffende Eigenschaft muß bei der Vereinbarung angegeben werden:

Quelle-Senke-Attribut ::=
IN | OUT | INOUT

IN bedeutet, daß diese Datenstation Quelle für Daten ist; d.h. sie darf nur in solchen Datenübertragungsanweisungen auftreten, die diese Daten *in* den Arbeitsspeicher übertragen (z.B. digitale Eingaben, Tastatur).

Mit OUT spezifizierte Datenstationen dürfen nur als Senken für Ausgaben aus dem Arbeitsspeicher benutzt werden (z.B. Drucker).

Eine Datenstation mit dem Attribut INOUT läßt Datenübertragungen in beiden Richtungen zu (z.B. Platte).

Die Datenübertragungen erfolgen mit dem rechnerinternen Format der Daten oder mittels Wandlung zwischen rechnerinternem und externem Format. PEARL bietet hierfür drei verschiedene Arten von E/A-Anweisungen:

- Die READ/WRITE-Anweisungen für die Übertragung mit rechnerinternem Format (z.B. für Plattendateien, siehe 10.4).
- Die PUT/GET-Anweisungen für die Übertragung mit Umwandlung zwischen internem Format und der Darstellung in dem Zeichensatz, der auf der Datenstation zur Verfügung steht (z.B. für Druckerausgabe, siehe 10.5).
- Die TAKE/SEND-Anweisungen für die Übertragung von Prozeßdaten (siehe 10.7).

Die Datenübertragung zu oder von einer Datenstation kann nur mit einer der angegebenen Arten erfolgen.

Die Auswahl wird bei der Vereinbarung der Datenstation mittels des Klassenattributs getroffen, das angibt, welcher der drei Klassen die Daten angehören:

- Sollen die READ/WRITE-Anweisungen benutzt werden, d.h. nimmt die Datenstation Daten in rechnerinternem Format auf, so wird als Klassenattribut der Typ der zu übertragenden Daten angegeben, z.B. FIXED oder FLOAT (53) oder BIT (16) oder ALL (für unterschiedliche Typen).
- Werden die Daten auf der Datenstation alpha-numerisch dargestellt (Fall PUT/GET), so erhält die Datenstation das Klassenattribut ALPHIC.
- Datenstationen für Übertragungen mit den TAKE/SEND-Anweisungen haben das Klassenattribut BASIC.

Die allgemeine Form des Klassenattributs lautet:

Klassenattribut ::=
    ALPHIC | **BASIC** | Typ-der-Übertragungsdaten

Typ der Übertragungsdaten ::=
    ALL | einfacher-Typ | zusammengesetzter-Typ

einfacher-Typ ::=
    Typ-ganze-Zahl | Typ-Gleitpunktzahl | Typ-Bitkette |
    Typ-Zeichenkette | Typ-Uhrzeit | Typ-Dauer

zusammengesetzter-Typ ::=
    EA-Struktur | Bezeichner§für-neuen-Typ-aus-einfachen-Typen

EA-Struktur ::=
    STRUCT **[** EA-Strukturkomponente [, EA-Strukturkomponente] ··· **]**

```
EA-Strukturkomponente :=
        Bezeichner-Angabe
        { einfacher-Typ | EA-Struktur | Bezeichner§für-neuen-Typ-aus-einfachen-Typen }
```

Der Typ der Übertragungsdaten darf also auch eine mehrfach strukturierte Struktur oder ein neu vereinbarter Typ sein, wobei jedoch keine Komponente vom Typ Referenz sein darf.

Beispiel:

```
...
TYPE    Art_Struktur   STRUCT
        [  (Nr, Anzahl ) FIXED ,
           Gewicht   FLOAT , ... ] ;
DCL     Art_Datei   DATION   INOUT Art_Struktur   ... ,
        Tab   DATION   INOUT   FIXED   ... ;
```

Die Angabe ALL schließt alle anderen Möglichkeiten von Typ-der-Übertragungsdaten ein.

Beispiel:

Auf einem Plattenspeichergerät mit dem Systemnamen PSP31 und dem Benutzernamen Platte sollen eine Datei File1 für die Eingabe von FIXED-Größen und eine Datei File2 für die Ein- und Ausgabe von FLOAT-Größen mit rechnerinternem Format kreiert werden.

```
SYSTEM ;
        Platte: PSP31 ;

PROBLEM ;
        SPC  Platte  DATION  INOUT  ALL  ... ;
        DCL  File1  DATION  IN  FIXED  ...  CREATED  (Platte) ;
        DCL  File2  DATION  INOUT  FLOAT  ...  CREATED  (Platte) ;
```

Das Quelle-Senke-Attribut und Klassenattribut müssen für jede Datenstation vereinbart werden. Nicht so die nun beschriebenen Attribute für die Gliederung und die Zugriffsmöglichkeiten einer Datenstation.

Die kleinste Datenmenge, die von oder zu einer Datenstation übertragen wird, heißt Datenelement. Ihr Typ ist durch das Klassenattribut bestimmt. Mehrere Datenelemente können zu einem Satz (synonym Zeile) und mehrere Sätze zu einem Abschnitt (synonym Seite) zusammengefaßt werden, d.h. die Gesamtheit der Elemente bildet ein 1-, 2- oder 3-dimensionales Feld. Dazu ist im Gliederungsattribut die Anzahl der Datenelemente einer Zeile, die Anzahl der Zeilen einer Seite und die Anzahl der Seiten mitzuteilen:

Gliederung ::=
  DIM ({ * **|** pgz } [, pgz [, pgz]]) [**TFU** [**MAX**]]

Dabei bezeichnet die jeweils am weitesten rechts stehende pgz-Angabe immer die Anzahl der Elemente je Zeile, die nächste (gegebenenfalls fehlende) pgz-Angabe die Anzahl der Zeilen je Seite und die dann folgende (gegebenenfalls fehlende) pgz-Angabe die Anzahl der Seiten. Die Angabe * bedeutet, daß die entsprechende Anzahl nicht begrenzt ist. Beispielsweise kann eine Datenstation Drucker mit 120 Zeichen je Zeile, 60 Zeilen je Seite und beliebig vielen Seiten die Gliederung (*, 60, 120) erhalten.

Möglich sind die folgenden Kombinationen:

- 3-dimensionale Gliederung

  DIM (Anzahl Seiten, Anzahl Zeilen, Anzahl Elemente) oder
  DIM (*, Anzahl Zeilen, Anzahl Elemente)

- 2-dimensionale Gliederung

  DIM (Anzahl Zeilen, Anzahl Elemente) oder
  DIM (*, Anzahl Elemente)

- 1-dimensionale Gliederung

  DIM (Anzahl Elemente) oder
  DIM (*)

Die Gliederung gibt zudem an, wieviele Datenelemente bei Ausführung *einer* Datenübertragungsanweisung *mindestens* übertragen werden:

- Fehlen die Angaben TFU und MAX, so können einzelne Datenelemente, Zeilen oder Seiten übertragen werden.

- Die Angabe TFU bedeutet, daß nur Zeilen oder Seiten übertragbar sind.

  Ist die aktuelle Anzahl der Datenelemente in einer Datenübertragungs-anweisung kleiner als die Anzahl der Datenelemente einer Zeile bzw. Seite, so wird diese Zeile bzw. Seite implizit mit Leerzeichen (ALPHIC-Datenstationen) oder Nullen (BASIC-Datenstationen) aufgefüllt.

  Beispiel:

  DCL Drucker **DATION OUT ALPHIC DIM**(*, 60, 120) **TFU** ... ;
  ...
  PUT 'PEARL' **TO** Drucker ;

  Diese PUT-Anweisung bewirkt, daß die fünf Zeichen P, E, A, R, L in einer (neuen) Zeile von Drucker ausgegeben werden.

- TFU MAX entspricht TFU

Die möglichen Zugriffsarten zu einer Datenstation bestimmt das Zugriffsattribut:

Zugriffsattribut ::=
{ **DIRECT** | **FORWARD** | **FORBACK** } [**NOCYCL** | **CYCLIC**]
[**STREAM** | **NOSTREAM**]

DIRECT bedeutet, daß (ausgehend von einem übertragenen Datenelement) auf ein beliebiges Datenelement unter Angabe der Position des Elements (siehe 10.4, 10.5) direkt zugegriffen werden kann.

Die Attribute FORWARD und FORBACK bedeuten sequentiellen Zugriff; d.h. der Zugriff darf (ausgehend von einem übertragenen Element) nur in der durch die Gliederung festgelegten Reihenfolge erfolgen - und zwar bei FORWARD nur vorwärts, bei FORBACK in beiden Richtungen - eventuell unter Angabe der relativen Position des gesuchten Elements zu dem gerade übertragenen Element (siehe 10.4, 10.5).

NOCYCL, CYCLIC, STREAM und NOSTREAM werden im Zusammenhang mit den E/A-Anweisungen in 10.4 und 10.5 behandelt.

Das Kontroll-Attribut ist aus Kompatibilitätsgründen syntaktisch zugelassen, hat aber keine semantische Bedeutung.

Kontroll-Attribut ::=
CONTROL (**ALL**)

Beispiel:

Auf einem Plattenspeichergerät mit dem Systemnamen PSP31 und dem Benutzernamen PLATTE soll eine Datei für eine Tabelle mit 300 Zeilen und 5 Spalten (Elemente pro Zeile) kreiert werden. Die Tabellenelemente seien Gleitpunktzahlen; der Zugriff soll elementweise direkt und nur lesend erfolgen.

```
MODULE ;

SYSTEM ;
    Platte: PSP31 ;
    ...
PROBLEM ;
    SPC  Platte  DATION  INOUT  ALL ;
    DCL  Tabelle  DATION  IN  FLOAT  DIM(300, 5)  DIRECT
         GLOBAL  CREATED (Platte) ;
    ...
```

Beispiele für die Vereinbarung von Datenstationen für die zeichenweise Ein- und Ausgaben sind in 10.5 und 10.4 beschrieben.

10.3 Öffnen und Schließen von Datenstationen (OPEN, CLOSE)

Bevor eine Datenstation zum ersten Mal in einer Datenübertragungsanweisung benutzt werden darf, muß sie mit der Open-Anweisung eröffnet werden:

```
Open-Anweisung ::=
    OPEN  Name§Dation  [ BY  Open-Parameter [ , Open-Parameter ] ··· ] ;
```

Durch Ausführung der Open-Anweisung wird eine Datenstation mit Gliederung auf ihren Anfang positioniert.

Die Open-Parameter dienen zum Umgang mit Datenstationen, die identifizierbare Datenbestände (Dateien) enthalten. Z.B. kann eine system-definierte Datenstation Platte einen Datenbestand TAB1 besitzen, der unter diesem Namen auch nach Ende des Programms verwahrt wird. Das gleiche oder ein anderes Programm kann zu einem späteren Zeitpunkt auf Platte eine benutzerdefinierte Datenstation Tabelle kreieren, die in der Open-Anweisung mit dem Datenbestand TAB1 identifiziert wird.

Open-Parameter ::=
IDF ({ Name§Character-Variable | Zeichenkettenkonstante }) |
RST (Name§Fehlervariable-Fixed) |
{ OLD | NEW | ANY } |
{ CAN | PRM }

Die Open-Parameter der Open-Anweisung müssen zu unterschiedlichen Untermengen gehören.

Bedeutung der Parameter:

- IDF (Name§Character-Variable | Zeichenkettenkonstante)
  Der Wert der angegebenen Character-Variablen oder die angegebenen Zeichenkettenkonstante sind der Name des Datenbestandes, der mit der unter Name§Dation aufgeführten Datenstation identifiziert werden soll.

- RST (Name§Fehlervariable-FIXED)
  Tritt während der OPEN-Ausführung ein Fehler auf, so wird die angegebene Variable mit einer Fehlernummer ungleich Null belegt; im fehlerfreien Fall wird sie auf Null gesetzt (siehe 10.8).

- OLD
  Existiert ein Datenbestand mit dem IDF-Namen, so wird er der angegebenen Datenstation zugeordnet. Anderenfalls, oder wenn IDF fehlt, erfolgt eine Fehlermeldung bzw. Setzen der RST-Variablen mit der Fehlernummer.

- NEW
  Es wird ein Datenbestand mit dem IDF-Namen angelegt und mit der angegebenen Datenstation identifiziert. Ist bereits ein Datenbestand mit diesem Namen vorhanden oder fehlt IDF, so erfolgt eine Fehlermeldung bzw. Setzen der RST-Variablen mit der Fehlernummer.

- ANY
  Existiert bereits ein Datenbestand mit dem IDF-Namen, so wird dieser mit der angegebenen Datenstation identifiziert. Anderenfalls wird hierfür ein neuer Datenbestand angelegt. Fehlt IDF, so wird ein neuer Datenbestand unter einem vom System bestimmten Namen angelegt und mit der angegebenen Datenstation identifiziert.

- CAN (von "cancel")
  Der Datenbestand ist nach Ausführung der Close-Anweisung (siehe unten) nicht mehr zugänglich zu machen.

- PRM (von "permanent")
  Der Datenbestand ist nach Ausführung der Close-Anwesiung noch vorhanden und nach erneuter Ausführung einer Open-Anweisung mit demselben IDF-Namen wieder zugänglich.

Bei fehlenden Open-Parametern werden ANY und PRM angenommen.

Durch die Ausführung der Close-Anweisung wird eine Datenstation geschlossen; d.h. sie ist dann erst wieder nach Ausführung einer Open-Anweisung benutzbar.

```
Close-Anweisung ::=
        CLOSE  Name§Dation  [  BY Close-Parameter [ , Close-Parameter ] ··· ] ;
```

Die in der Open-Anweisung getroffenen Einstellungen für das Schließen einer Datenstation können durch eine Close-Anweisung überschrieben werden:

```
Close-Parameter ::=
        CAN | PRM | RST (Name§Fehlervariable-FIXED)
```

Allgemein gelten folgende Regeln:

- Nicht jede Task, die einen Zugriff auf eine Datenstation ausführt, muß eine Open- oder Close-Anweisung ausführen.
- Es muß jedoch bezüglich des Zugriffs auf eine Datenstation mindestens eine Open-Anweisung durchlaufen werden.
- Es müssen gleichviele Close-Anweisungen wie Open-Anweisungen durchgeführt werden, damit die Datenstation geschlossen ist.
- Sich entsprechende Open- und Close-Anweisungen müssen nicht durch die gleiche Task ausgeführt werden.
- Bei fehlenden Parametern werden ANY und PRM angenommen, sofern nicht vorher ausgeführte Close- oder Open-Anweisungen explizite Einstellungen vorgenommen haben.

Beispiel:

```
MODULE ;
SYSTEM ;
        Drucker : DRUA ;
        Platte : PSP31 ;
        ...
PROBLEM ;
        SPC    Drucker  DATION  OUT  ALPHIC ;
        DCL    Tab_Prot  DATION OUT  ALPHIC  DIM(*, 50, 30)  FORWARD
               GLOBAL  CREATED (Drucker) ;
        SPC    Platte  DATION  INOUT  ALL ;
        DCL    Tabelle  DATION  IN  FLOAT  DIM(300, 5)  DIRECT
               GLOBAL  CREATED (Platte) ;

Start : TASK  MAIN ;
        OPEN  Tab_Prot ;
        OPEN  Tabelle  BY  IDF ('TAB-1'), OLD ;
        ACTIVATE  Prot ;
        ...
        END ;  ! Start

Prot :  TASK ;
        ! Datenübertragungsanweisungen mit Tabelle und Tab_Prot
        ...
        CLOSE  Tab_Prot ;
        CLOSE  Tabelle ;
        END ;  ! Prot
...
MODEND ;
```

10.4 Die Read- und die Write-Anweisung (READ, WRITE)

Die Read-Anweisung dient zur Eingabe, die Write-Anweisung zur Ausgabe von Daten ohne Wandlung der rechnerinternen Darstellung (binäre Ein- und Ausgabe). Es können Daten von oder zu den angeschlossenen Geräten (z.B. eine Datei auf Platte oder Magnetband) übertragen werden. Die entsprechenden Datenstationen müssen mit dem Klassenattribut "Typ-der-Übertragungsdaten" vereinbart worden sein.

Beispiele:

(1) Die Spalten 4 und 5 in der Gliederung Tabelle (vgl. das Beispiel auf Seite 10-9) sollen durch neu berechnete Werte ersetzt werden.

```
...
DCL   (x, y, z)   FLOAT ;
...
FOR  Zeile  FROM  1  TO  300
REPEAT
        ! Berechnung von x, y, z
        WRITE  x, SIN(y+z)  TO  Tabelle  BY  POS (Zeile, 4) ;
END ;
```

(2) Eine Task Messung holt periodisch 14 Temperaturwerte (Procedure Get_Temp), verarbeitet sie und schreibt sie sequentiell in Blöcken zu 14 Werten in ein Logbuch auf Platte.

```
SYSTEM ;
        File : DISC ;

PROBLEM ;
        SPC     Get_Temp  PROC (i FIXED)  RETURNS  ( FIXED (15) ) GLOBAL;
        SPC     File  DATION  INOUT  ALL ;
        DCL     Logbuch  DATION  OUT  FIXED(15)  DIM(*, 14)  TFU  FORWARD
                CREATED  (File) ;
        DCL     Anz_Temp  INV  FIXED (15)  INIT (14) ;

Start:  TASK  MAIN ;
        OPEN  Logbuch ;       ! Positionierung auf den Anfang
        ALL  10 SEC  ACTIVATE  Messung ;
        ...
        END ;  ! Start

Messung:  TASK ;
        DCL  Temperatur (Anz_Temp)  FIXED (15) ;
        FOR  i  TO  Anz_Temp
        REPEAT
                Temperatur (i)  :=  Get_Temp (i);
        END ;
        ! Verarbeitung der Messwerte
        WRITE  Temperatur  TO  Logbuch ;
        END ;  ! Messung
```

Die allgemeinen Formen der Read- und Write-Anweisungen lauten:

Read-Anweisung ::=
READ [{ Name§Variable **|** Ausschnitt } [, { Name§Variable **|** Ausschnitt }] ···]
FROM Name§Dation [**BY** Position [, Position] ···] ;

Write-Anweisung ::=
WRITE [{ Ausdruck **|** Ausschnitt } [, { Ausdruck **|** Ausschnitt }] ···]
TO Name§Dation [**BY** Position [, Position] ···] ;

Ausschnitt ::=
Name§Feld ([Index ,] ··· Index **:** Index)

Index ::=
Ausdruck§mit-ganzer-Zahl-als-Wert

Position ::=
absolute-Position **|** relative-Position **|** **RST** (Name§Fehlervariable-FIXED)

absolute-Position ::=
{ **COL** **|** **LINE** } (Ausdruck) **|**
POS ([[Ausdruck ,] Ausdruck ,] Ausdruck) **|**
SOP ([[Name ,] Name ,] Name)

relative-Position ::=
{ **X** **|** **SKIP** **|** **PAGE** } [(Ausdruck)] **|**
ADV ([[Ausdruck ,] Ausdruck ,] Ausdruck)

Bei der Eingabe mit der Read-Anweisung werden die angesteuerten Datenelemente nacheinander gelesen und den Variablen in der Variablenliste korrespondierend zugewiesen. Die Zuweisung zu den Variablen erfolgt dabei nach den allgemeinen Regeln für Zuweisungen. Ist ein Element der Variablenliste ein Feld, so werden die angesteuerten Daten zeilenweise zugewiesen; ist es eine Struktur, so werden die Daten den Strukturkomponenten in der durch die Strukturvereinbarung festgelegten Reihenfolge zugewiesen.

Der einfacheren Schreibweise wegen können in der Variablenliste aufeinanderfolgende Elemente (der letzten Dimension) eines Feldes in Form eines Ausschnitts angegeben werden. Sei etwa Liste ein Feld mit zehn Elementen Liste(1), ... , Liste(10); dann sind die beiden folgenden Anweisungen äquivalent:

READ Liste(2), Liste(3), Liste(4) **...** ;
READ Liste(2 **:** 4) **...** ;

Alle Positionsangaben werden vollständig ausgewertet, bevor den Datenelementen Werte zugewiesen werden.

Beispiel:

> Lies X von Position 3 und Y von Position 5 aus der Dation File ist gleich den Anweisungen:
>
> **READ** X **FROM** File **BY** **POS**(3) ;
> **READ** Y **FROM** File **BY** **POS**(5) ;
>
> aber nicht der Anweisung
>
> **READ** X, Y **FROM** File **BY** **POS**(3), **POS**(5) ;
>
> Die Ausführung dieses Statements hat zur Folge, daß X von der Position 5 und Y von der darauf folgenden Position gelesen wird.

Die RST-Angabe (vgl. 10.8) kann an beliebiger Stelle in der Positionsliste stehen. Sie wirkt allerdings erst dann, wenn sie ausgewertet wurde. Eine Positionsliste wird, von links beginnend, der Reihe nach abgearbeitet. Tritt hierbei ein Fehler auf, so wird die Abarbeitung der E/A-Anweisung an dieser Stelle abgebrochen und die zu diesem Zeitpunkt gültige Fehlerreaktion (Fehlerzuweisung an eine RST-Variable oder Systemreaktion) durchgeführt.

Diese Aussagen gelten analog für die Write-Anweisung.

Der Typ der Variablen in der Variablenliste der Read-Anweisung muß mit dem Klassenattribut der angegebenen Datenstation verträglich sein; dies gilt analog für die Ergebnisse der Ausdrücke in der Ausdrucksliste der Write-Anweisung.

Die Variablen- oder Ausdrucksliste der Read- oder Write-Anweisung kann fehlen, wenn diese Anweisungen nur zum Positionieren in der angegebenen Datenstation benutzt werden sollen. Dann muß allerdings eine Position angegeben sein.

Die Angaben in der Positionsliste beziehen sich auf die Gliederung der Datenstation und bestimmen die Datenelemente, die übertragen werden sollen. Deshalb müssen die Werte der Ausdrücke vom Typ FIXED und mit der Gliederung verträglich sein.

Bei Benutzung einer absoluten, d.h. vom aktuellen Datenelement unabhängigen Position muß die Datenstation das Zugriffsattribut DIRECT besitzen. Eine relative Position bezeichnet den Abstand des zu übertragenden Datenelements vom aktuellen Datenelement; in diesem Fall muß die Datenstation das Zugriffsattribut FORWARD, FORBACK oder DIRECT haben.

Im einzelnen haben die möglichen Positionsangaben folgende Bedeutung:

- COL (Ausdruck)

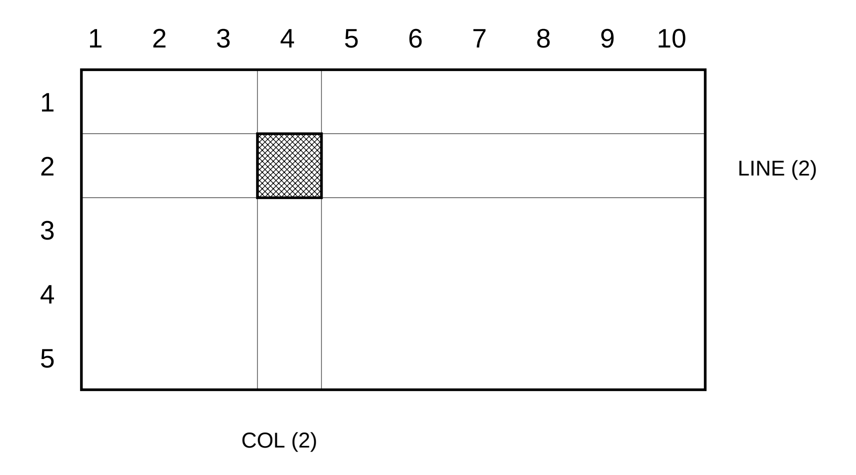
  bezieht sich auf die erste Dimension (von rechts) der Gliederung und bestimmt das i-te Element in der aktuellen Zeile der Datenstation, wenn i gleich dem Wert des Ausdrucks ist.

- LINE (Ausdruck)

  bezieht sich auf die zweite Dimension der Gliederung und bestimmt die i-te Zeile in der aktuellen Seite der Datenstation, wenn i gleich dem Wert des Ausdrucks ist.

Beispiel:

Der Gliederung (5, 10) entspricht ein 2-dimensionales Feld:



- POS ([[Ausdruck§Seite ,] Ausdruck§Zeile ,] Ausdruck§Spalte)

  gibt die Position eines Datenelements in der n-dimensionalen Gliederung (n = 1, 2, 3) einer Datenstation an. Fehlende Ausdrücke werden durch den jeweiligen aktuellen Wert ersetzt.

Beispiel:

Durch die Anweisung

READ x **FROM** File1 **BY** **POS** (3,2,8) ;

wird das achte Datenelement der zweiten Zeile der dritten Seite von File1 nach x eingelesen. Folgt anschließend die Anweisung

READ x **FROM** File1 **BY** **POS** (4,5) ;

so wird das fünfte Datenelement der vierten Zeile der dritten Seite von File1 nach x eingelesen.

- SOP ([[Name§Seite ,] Name§Zeile ,] Name§Spalte)

  ist das Gegenstück zu POS. Mit SOP können die aktuellen Positionen einer Dation den angegebenen Programmvariablen zugewiesen werden. Es dürfen nicht mehr Namen in SOP angegeben werden als die angesprochene Dation Dimensionen hat.

Im folgenden stehe i für den Wert des jeweiligen Ausdrucks.

- X [(Ausdruck)]

  bezieht sich auf die erste Dimension der Gliederung und bestimmt das i-te Element nach (i positiv) bzw. vor (i negativ) dem aktuellen Element in der aktuellen Zeile der Datenstation; i = 0 bezeichnet das aktuelle Element.

- SKIP [(Ausdruck)]

  bezieht sich auf die zweite Dimension der Gliederung und bestimmt den Anfang (das erste Element) der i-ten Zeile nach (i positiv) bzw. vor (i negativ) der aktuellen Zeile in der aktuellen Seite der Datenstation; i = 0 bezeichnet die aktuelle Zeile.

- PAGE [(Ausdruck)]

  bezieht sich auf die dritte Dimension der Gliederung und bestimmt den Anfang der i-ten Seite nach (i positiv) bzw. vor (i negativ) der aktuellen Seite der Datenstation; i = 0 bezeichnet die aktuelle Seite.

Fehlt die Ausdrucksangabe bei X, SKIP bzw. PAGE, so wird der Wert 1 angenommen. Der Wert eines angegebenen Ausdrucks muß positiv sein, wenn die Datenstation das Attribut FORWARD besitzt.

- ADV ([[Ausdruck§Seite ,] Ausdruck§Zeile ,] Ausdruck§Spalte)

  gibt den Abstand des zu übertragenden Datenelements von dem aktuellen Datenelement an. Fehlende Ausdrücke werden durch den Wert Null ersetzt. Hat die Datenstation das Attribut FORWARD, so muß der Wert des am weitesten links angegebenen Ausdrucks positiv sein.

Beispiele:

Die betreffende Datenstation File1 habe die Gliederung (10, 10, 10), die aktuelle Position sei z.B. (5, 3, 8).

Positionsangabe	neue Position	
X	(5,3,9)	
X (-5)	(5,3,3)	
X (4)	(5,4,2)	(1)
SKIP (2)	(5,5,1)	
SKIP (-1)	(5,2,1)	
PAGE	(6,1,1)	
PAGE (6)	(1,1,1)	(2)
PAGE (-4)	(1,1,1)	
ADV (2,5,1)	(7,8,9)	
ADV (1,0)	(5,4,8)	
ADV (-3,-2,1)	(2,1,9)	
ADV (1,8,0)	(7,1,8)	(3)

Bei einer relativen Positionierung dürfen Dimensionsgrenzen nicht überschritten werden (vgl. die gekennzeichneten Beispiele), sofern die Datenstation nicht das Zugriffsattribut STREAM bzw. CYCLIC besitzt. STREAM erlaubt die Überschreitung der inneren Dimensionsgrenze (Beispiel (1) und (3)), nicht aber, die Grenze der höchsten Dimension zu überschreiten (Beispiel (2)). Hierfür muß eine Datenstation das Attribut CYCLIC besitzen. Sollen bei den entsprechenden Grenzüberschreitungen Fehlerreaktionen erfolgen, müssen die Attribute NOSTREAM bzw. NOCYCL angegeben werden. Standardmäßig werden STREAM und NOCYCL angenommen.

Beispiel:

DCL File1 **DATION INOUT FIXED DIM**(10,10,10) **FORWARD CYCLIC**
GLOBAL CREATED (Platte) ;

Die aktuelle Position sei z.B. jeweils (5,3,8). Dann gilt:

Positionsangabe	neue Position
X (6)	(5,4,4)
SKIP (8)	(6,1,1)
PAGE (7)	(2,1,1)
ADV (9,0,3)	(4,4,1)

10.5 Die Get- und Put-Anweisung (GET, PUT)

Die Get-Anweisung dient zur Eingabe, die Put-Anweisung zur Ausgabe von Daten mit Wandlung zwischen rechnerinterner und externer, zeichenorientierten Darstellung auf ALPHIC-Datenstationen. Zur Steuerung dieser Wandlung können Formate angegeben werden.

Beispiele:

(1) Auf dem Monitor eines Lagerverwalters soll folgender Text erscheinen:

```
___Artikel-Nr:__4711
___Bestand:_____1281
```

Nötige Programmschritte:

```
...
SPC   Monitor  DATION  INOUT  ALPHIC ;
DCL   Lager_Monitor  DATION  INOUT  ALPHIC  DIM(*,20,80)
      FORWARD  GLOBAL  CREATED (Monitor) ;
DCL  (Artnr, Best)  FIXED ;
...
PUT  'Artikel-Nr:',  Artnr,  'Bestand:',  Best  TO  Lager_Monitor
      BY  X(3),  A(13),  F(4),  SKIP,  X(3),  A(13),  F(4) ;
...
```

(2) Ausgabe von zwei Werten im Standardformat auf einer neuen Seite des Druckers.

```
...
SPC   Thermo_Print  DATION  OUT  ALPHIC ;
DCL   Drucker  DATION  OUT  ALPHIC  DIM(*,50,120)  FORWARD
      GLOBAL  CREATED (Thermo_Print) ;
DCL  a  FIXED (15),  x  FLOAT (31) ;
...
a := 5 ;
x := 2.33 ;
...
PUT  TO  Drucker  BY  PAGE ;
PUT  a, x  TO  Drucker  BY  LIST ;
```

Die Ausführung ergibt folgendes Druckbild:

```
_____5___2.33000E+00
```

(3) In einer Eingabedatei auf Diskette seien Daten in folgender Form abgelegt:

Spalte 1 - 10 : Artikelbezeichnung (CHARACTER)
Spalte 12 - 20 : Menge (FIXED)
Spalte 22 - 30 : Preis je Einheit rechtsbündig (z.B. ___ 124.57)

Sie sollen in die Variablen Art_Bez, Menge, Preis eingelesen werden:

...
SPC Diskette **DATION INOUT ALL** ;
DCL Eingabedatei **DATION IN ALPHIC DIM**(*,80) **TFU FORWARD**
GLOBAL CREATED (Diskette),
Art_Bez **CHAR** (10),
Menge **FIXED**,
Preis **FLOAT** ;
...
GET Art_Bez, Menge, Preis **FROM** Eingabedatei **BY**
A(10), **X**, **F**(9), **X**, **E**(9), **SKIP** ;

Die allgemeinen Formen der Get- und Put-Anweisungen lauten:

Get-Anweisung ::=
GET [{ Name§Variable | Ausschnitt } [, { Name§Variable | Ausschnitt }] ···]
FROM Name§Dation [**BY** Format-Position [, Format-Position] ···] ;

Put-Anweisung ::=
PUT [{ Ausdruck | Ausschnitt } [, { Ausdruck | Ausschnitt }] ···]
TO Name§Dation [**BY** Format-Position [, Format-Position] ···] ;

Format-Position ::=
[Faktor] { Format | Position } |
Faktor (Format-Position [, Format-Position] ···)

Faktor ::=
(Ausdruck§ganze-Zahl-größer-Null) | ganze-Zahl-ohne-Genauigkeit§größer-Null

Format ::=
Fixed-Format | Float-Format | Zeichenketten-Formate | Bit-Format |
Zeit-Format | Dauer-Format | List-Format | R-Format | **RST** (Name)

Bei der Eingabe mit der Get-Anweisung werden die angesteuerten Datenelemente nacheinander gelesen und den Variablen in der Variablenliste korrespondierend

(analog zur Read-Anweisung) zugewiesen. Die Zuweisung zu den Variablen erfolgt dabei nach den allgemeinen Regeln für Zuweisungen.

Die Eingabe ist beendet, wenn die Variablenliste abgearbeitet ist. Sind noch Listen-elemente vorhanden, aber keine Datenelemente mehr, wird eine Fehlermeldung ausgegeben.

Bei Ausführung der Put-Anweisung werden die Werte der in der Liste nach PUT angegebenen Ausdrücke in der aufgeführten Reihenfolge ausgegeben.

In der Reihenfolge der Niederschrift ist jeder Variablen in der Variablenliste der Get-Anweisung ein Format in der Format-Positions-Liste zugeordnet, das die externe Darstellung der Daten auf der angegebenen Datenstation beschreibt und zur Wand-lung in die rechnerinterne Darstellung benutzt wird. Die Art des Formats wird durch den Typ der Variablen bestimmt. Neben Formaten kann die Liste der Format-Positio-nen auch Positionsangaben (vgl. 10.4) zur Positionierung in der Datenstation enthal-ten. Ist die Variablenliste noch nicht erschöpft, so werden die anstehenden Positio-nierungen ausgeführt und die eben genannte Zuordnung erst mit dem nächst fol-genden Format fortgesetzt. Ist die Variablenliste dagegen schon erschöpft, so wer-den die nachfolgenden Positionsangaben ausgewertet, bis ein Format auftritt oder die Liste abgearbeitet ist.

Enthält die Variablenliste ein Feld oder Ausschnitt, so werden die folgenden Formate den Feldelementen nacheinander zugeordnet.

Die Anzahl der übertragenen Datenelemente wird nur durch die Variablenliste bestimmt, nicht durch die Format-Positions-Liste. Sind mehr Formate als Variablen vorhanden, so werden die überzähligen Formate ignoriert. Sind noch Variablen vorhanden, wenn die Format-Positionsliste schon abgearbeitet ist, so wird wieder mit dem ersten Element der Format-Positionsliste begonnen. In jedem Fall ist die Über-tragung beendet, wenn die Variablenliste erschöpft ist.

Diese Ausführungen gelten analog für die Put-Anweisung, wenn "Variable" durch "Ausdruck" ersetzt wird.

Die Datenstation muß das Klassenattribut ALPHIC, eine Gliederung und ein Zugriffs-attribut besitzen. Die Wahl des Zugriffsattributs schränkt evtl. die Positionierungs-möglichkeiten ein (vgl. 10.4).

Die Format-Positionsliste besteht aus Format- und Positionsangaben. Zur Verein-fachung der Schreibweise dürfen in der Liste Wiederholungsfaktoren benutzt werden. Beispielsweise läßt sich die Format-Positionsliste

```
X(2),  F(12,3),  X(2),  F(12,3),  X(2),  F(12,3)
```

einfacher so schreiben:

(3) (X(2), F(12,3))

Die folgende Tabelle zeigt die zugelassenen Zuordnungen zwischen Formaten und den Typen der zu übertragenden Datenelemente:

Format	Datentyp
Fixed-Format	FIXED, FLOAT
Float-Format	FIXED, FLOAT
Bit-Format	BIT
Zeichenketten-Format	CHARACTER
Zeit-Dauer	CLOCK
Dauer-Format	DURATION
List-Format	alle angegebenen Datentypen

Im einzelnen haben die Formate folgende Form und Bedeutung (die Positionsangaben wurden in 10.4 erklärt):

10.5.1 Das Fixed-Format (F)

Fixed-Format ::=
    F (Feldweite [, Dezimalstellen [, Skalenfaktor]])

Feldweite ::=
    Ausdruck§mit-positiver-ganzer-Zahl-als Wert

Dezimalstellen ::=
    Ausdruck§mit-nicht-negativer-ganzer-Zahl-als-Wert

Skalenfaktor ::=
    Ausdruck§mit-ganzer-Zahl-als-Wert

Das Fixed-Format beschreibt die externe Darstellung dezimaler Festpunktzahlen. Die Feldweite w ist die Gesamtzahl der Zeichen, die der Dezimalzahl zur Verfügung steht; Dezimalstellen d bezeichnet die Anzahl der Ziffern hinter dem Dezimalpunkt. Der Skalenfaktor p kann sowohl positiv als auch negativ sein; er bewirkt, daß nicht die Zahl selbst, sondern ihr mit 10**p multiplizierter Wert übertragen wird.

Seine Bedeutung liegt darin, daß mit dem Fixed-Format zwar nur ganze Zahlen übertragen werden können, durch Skalierung jedoch auch die Verarbeitung gebrochener Festpunktzahlen möglich ist. Diese werden bei der Eingabe in ganze Zahlen und bei der Ausgabe wieder in gebrochene Zahlen umgewandelt.

(1) Ausgabe

(1.1) Die Dezimalzahl wird rechtsbündig in einem Feld der Länge w in der Form

[-] pgz [. pgz]

abgelegt, wobei pgz eine positive ganze Zahl bedeutet. Nimmt die Zahl nicht das ganze Feld ein, so wird der linke Teil mit Leerzeichen aufgefüllt.

(1.2) Ist 0 > d oder w < d, so wird w-mal das Zeichen * abgelegt.

(1.3) Im Fall w < = 0 wird kein Zeichen abgelegt; es wird ein Fehler gemeldet.

(1.4) Ist d = 0 oder nicht angegeben, so wird nur der ganzzahlige Anteil der Dezimalzahl ohne Dezimalpunkt gerundet ausgegeben.

(1.5) Mit Ausnahme der Null direkt vor dem Dezimalpunkt werden führende Nullen unterdrückt.

(2) Eingabe

(2.1) Es wird ein Feld der Länge w gelesen, das eine dezimale Festkommazahl in folgender Darstellung enthält:

[[+ | -] pgz [. [pgz]]]

(2.2) Leerzeichen, die der Zahl vorangehen oder nachfolgen, werden ignoriert.

(2.3) Ist das ganze Feld leer, wird der Wert 0 eingelesen.

(2.4) Tritt in der Darstellung kein Dezimalpunkt auf, so werden die letzten d Ziffern als Stellen nach einem Dezimalpunkt interpretiert. Es muß p > = d sein.

(2.5) Tritt in der Darstellung ein Dezimalpunkt vor den letzten b Ziffern auf, so hat dieser Vorrang vor der Spezifikation durch d. In diesem Fall ist die Angabe von d bedeutungslos. Es muß p > = b sein.

(2.6) Ist w < = 0, so erfolgt keine Zuweisung; es wird ein Fehler gemeldet.

Beispiel:

Wert	Format	Ausgabe	
13.5	F(7,2)	__13.50	
275.2	F(4,1)	****	Fehlermeldung!
22.8	F(5)	___23	
212.73	F(9,2,2)	_21273.00	
212.73	F(9,2)	___212.73	

10.5.2 Das Float-Format (E)

Float-Format ::=
E (Feldweite [, Dezimalstellen [, Signifikanz]])

Signifikanz ::=
Ausdruck§mit-ganzer-Zahl-als-Wert

Das Float-Format beschreibt die externe Darstellung dezimaler Gleitpunktzahlen der Form

[+ | -] Gleitpunktzahl

wobei der Exponent aus zwei Ziffern besteht. Feldweite und Dezimalstellen haben dieselbe Bedeutung wie im Fixed-Format; die Signifikanz s bezeichnet die Anzahl der signifikanten Ziffern, d.h. die Länge der Mantisse.

E (w,d) ist gleichwertig mit E(w,d,d+1), und E(w) ist gleichwertig mit E(w,0).

(1) Ausgabe

Die Gleitpunktzahl wird rechtsbündig in einem Feld der Länge w abgelegt. Im übrigen gilt (1.1) von 10.5.1.

Ist 0 < w < d, so erfolgt eine Fehlerreaktion.

Im Fall 0 < w > d > s wird die Mantisse so gewählt, daß gilt:

$10^{s-d-1} \leq |\, \text{Mantisse} \,| < 10^{s-d}$

Für w = 0 wird kein Zeichen abgelegt; der zugehörige Ausdruck in der Ausdrucksliste wird übergangen.

Ist d > 0, so hat die Zahl die Form

[-] s-d Ziffern . d Ziffern E { + | - } Exponent.

Der Exponent wird so bestimmt, daß die führende Ziffer der Mantisse ungleich Null ist, sofern die Zahl von Null verschieden ist.

Ist d = 0, so hat die Zahl die Form

[-] s Ziffern E { + | - } Exponent

Ist w zu klein, um eine Ziffer der Mantisse ausgeben zu können, wird w-mal das Zeichen * ausgegeben, gefolgt von einer Fehlerreaktion.

(2) Eingabe

Es wird ein Feld der Länge w gelesen, das eine dezimale Gleitpunktzahl in einer der möglichen Darstellungen (vgl. 5.3) enthält.

Die Aussagen (2.2) bis (2.6) von 10.5.1 gelten analog.

Beispiele:

Wert	Format	Ausgabe
-0.07	E(9,1)	__7.0E-02
2713.5	E(11,2,4)	__27.13E+02
2721	E(8)	___2E+03

10.5.3 Die Zeichenketten-Formate (A) und (S)

Zeichenketten-Format ::=
A [(Ausdruck§Anzahl-Zeichen)] | S (Name§Anzahl-Zeichen-Variable-Fixed)

Die Zeichenketten-Formate beschreiben die externe Darstellung von Zeichenketten (Character-Größen) der Form

Zeichen…

Zeichenketten-Format (A)

Der Wert des Ausdrucks im Zeichenketten-Format bedeutet die Gesamtzahl w der für die Darstellung verfügbaren Zeichenpositionen.

(1) Ausgabe

Hat das Format die Form "A (Ausdruck)", so wird die Zeichenkette in der oben dargestellten Form linksbündig in ein Feld der Länge w ausgegeben. Besteht sie aus mehr als w Zeichen, wird sie rechts abgeschnitten; besteht sie aus weniger als w Zeichen, wird das Feld rechts mit Leerzeichen aufgefüllt. Ist w = 0, so werden keine Zeichen ausgegeben und der Ausdruck in der Ausdrucksliste der Put-Anweisung übergangen.

Ist der Ausdruck im Format nicht angegeben, das Format hat also die Form "A", so wird die Kette in ein Feld ausgegeben, dessen Länge gleich der Kettenlänge ist.

(2) Eingabe

Es werden maximal w oder bis zum nächsten Satz-Terminator (z.B. cr) Zeichen eingelesen. Ein Satz-Terminator wird nicht in die Zeichenketten-variable übertragen.

Ist w kleiner als die Länge lg der zugehörigen Zeichenkettenvariablen, so wird rechts mit Leerzeichen aufgefüllt; im Fall w > lg wird rechts abgeschnitten. Ist w = 0, so erhält die Variable eine Kette von lg Leerzeichen zugewiesen.

Beispiele:

Die Ausgabe der Zeichenkette 'PEARL' im Format

- A ergibt PEARL
- A(5) ergibt PEARL
- A(7) ergibt PEARL__
- A(2) ergibt PE

Die Eingabe der Zeichenkette 'PEARL__' an eine CHAR(5)-Variable Text im Format

- A ist äquivalent zu Text := 'PEARL' ;
- A(5) ist äquivalent zu Text := 'PEARL' ;
- A(7) ist äquivalent zu Text := 'PEARL' ;
- A(2) ist äquivalent zu Text := 'PE _ _ _ ' ;

Zeichenketten-Format (S)

Die Variable im Zeichenketten-Format muß vom Typ FIXED sein.

(1) Ausgabe

Identisch zum A-Format: der Wert der angegebenen Variablen bestimmt die Breite des Ausgabefeldes.

(2) Eingabe

Es werden maximal lg (Länge der zugehörigen Zeichenkettenvariablen) oder bis zum nächsten Satz-Terminator Zeichen eingelesen; ansonsten gelten die gleichen Regeln wie beim A-Format. Zusätzlich wird die Anzahl gelesener Zeichen (ohne Satz-Terminator) der Variablen des S-Formats zugewiesen. Welche Satz-Terminatoren für welche Geräte definiert sind, ist dem jeweiligen PEARL-Benutzerhandbuch zu entnehmen.

Beispiel:

Es sollen Kommandozeilen vom Terminal eingelesen werden.

DCL buffer **CHAR** (80) ;
DCL length **FIXED** ;
...
GET buffer **FROM** terminal **BY** **S**(length) ;
...

Nach der Eingabe "abc <RETURN>" auf der Tastatur (die Taste <RETURN> erzeugt einen Satz-Terminator), enthält die Variable "buffer" die Zeichen "abc" und die Variable "length" den Wert 3.

10.5.4 Das Bit-Format (B)

Bit-Format ::=
{ **B** | **B**1 | **B**2 | **B**3 | **B**4 } [(Ausdruck§Anzahl-Zeichen)]

Das Bit-Format beschreibt die externe Darstellung von Bitketten (Bit-Größen) und zwar (vgl. 5.4)

- in binärer Form durch das Format: { B | B1 } [(Ausdruck)] ,
- in Form von Tetraden durch das Format: B2 [(Ausdruck)] ,
- in Form von Oktaden durch das Format: B3 [(Ausdruck)] ,
- in hexadezimaler Form durch das Format: B4 [(Ausdruck)] .

Der Wert des Ausdrucks im Bit-Format bedeutet die Gesamtzahl w der für die Darstellung verfügbaren Zeichenpositionen.

(1) Ausgabe

Ist der Ausdruck im Format angegeben, so wird die Bitkette in der oben dargestellten Form linksbündig in ein Feld der Länge w ausgegeben. Besteht sie aus mehr als w Zeichen, wird sie rechts abgeschnitten; besteht sie aus weniger als w Zeichen, wird das Feld rechts mit Nullen aufgefüllt. Ist w nicht angegeben und hat das Bit-Format also die Form B | B1 | B2 | B3 | B4, so wird die Kette in ein Feld ausgegeben, dessen Länge gleich der Kettenlänge ist.

(2) Eingabe

Der Ausdruck muß angegeben werden.

Es wird ein Feld der Länge w eingelesen, das eine Bitkette der oben beschriebenen Form enthalten muß. Das Feld darf nicht ausschließlich aus Leerzeichen bestehen. Der Kette vorangehende oder nachfolgende Leerzeichen werden ignoriert. Die Aussagen zur Eingabe mit dem A-Format gelten sinngemäß.

Beispiele:

Die Ausgabe der Bitkette '0101110' im Format

- B(5) ergibt 01011
- B2(3) ergibt 113
- B3(3) ergibt 270
- B4(2) ergibt 5C

Die Variable Bitkette sei vom Typ BIT(8); für die Eingabe ergeben sich etwa folgende Möglichkeiten:

einzugebendes Datenelement	Format	Wert von Bitkette
11111	B(5)	11111000
201	B2(3)	10000100
235	B3(3)	01001110
AB	B4(2)	10101011

10.5.5 Das Zeit-Format (T)

Zeit-Format ::=
T (Feldweite [, Dezimalstellen])

Das Zeit-Format beschreibt die externe Darstellung von Uhrzeitangaben. Die Feldweite bedeutet die Gesamtzahl w der für die Darstellung verfügbaren Zeichenpositionen; Dezimalstellen steht für die Anzahl d der Ziffern für die Sekundenbruchteile der Uhrzeit.

(1) Ausgabe

Die Uhrzeit wird rechtsbündig in einem Feld der Länge w in der Form

[Ziffer] Ziffer : Ziffer Ziffer : Ziffer Ziffer [. pgz]

ausgegeben. Ist die erste Ziffer eine Null, wird sie durch ein Leerzeichen ersetzt. Im Fall d = 0 werden Dezimalpunkt und Sekundenbruchteile nicht mit ausgegeben.

Nimmt der Ausgabewert nicht das ganze Feld ein, so wird der linke Teil mit Leerzeichen aufgefüllt.

(2) Eingabe

Es wird ein Feld der Länge w eingelesen, das eine Uhrzeit in einer erlaubten Darstellung enthalten muß (siehe (1)). Vorangehende oder nachfolgende Leerzeichen werden ignoriert.

Beispiele:

Wert	Format	Ausgabe
12.30 Uhr 5.2 Sek	T(12,1)	__12:30:05.2
8 Uhr	T(8)	_8:00:00

10.5.6 Das Dauer-Format (D)

Dauer-Format ::=
D (Feldweite [, Dezimalstellen])

Das Dauer-Format beschreibt die externe Darstellung von Zeitdauern. Der Wert der Feldweite bedeutet die Gesamtzahl w der für die Darstellung verfügbare Zeichenpositionen, der Wert von Dezimalstellen die Anzahl d der Ziffern für die Sekundenbruchteile der Dauer.

(1) Ausgabe

Die Dauer wird rechtsbündig in einem Feld der Länge w in der Form

[Ziffer] Ziffer_HRS_Ziffer Ziffer_MIN_Ziffer Ziffer [. pgz]_SEC

ausgegeben. Es gelten die Regeln von 10.5.5 (1).

(2) Eingabe

Es wird ein Feld der Länge w eingelesen, das eine Dauer in einer erlaubten Darstellung enthalten muß (siehe (1)). Vorangehende oder nachfolgende Leerzeichen werden ignoriert.

Wert	Format	Ausgabe
11 Stunden 15 Minuten	D(20)	11_HRS_15_MIN_00_SEC
100 Millisekunden	D(24,3)	_0_HRS_00_MIN_00.100_SEC

Dabei bedeutet _ ein Leerzeichen.

10.5.7 Das List-Format (LIST)

List-Format ::=
LIST

Das List-Format dient zur Ein- und Ausgabe von Fixed-, Float-, Bit-, Char-, Clock- und Dur-Größen.

(1) Ausgabe

Aufeinanderfolgende Ausgabedaten werden durch je zwei Zwischenräume getrennt. Die Daten werden so ausgegeben, als ob für eine Größe vom Typ

CHAR(k)	das Format	A(k),
BIT(k)	"	B(k),
FIXED(k)	"	F(n),
FLOAT(k)	"	E(m,m-7,m-6),
CLOCK	"	T(8),
DUR	"	D(20)

mit n = ENTIER (k/3.32) + 2, m = ENTIER (k/3.32) + 8 vereinbart wäre.

(2) Eingabe

Die Eingabedaten können irgendeine Form haben, die für die Darstellung von Konstanten erlaubt ist. Sie werden durch ein Komma oder mindestens zwei Zwischenräume getrennt. Steht zwischen zwei Kommas keine Konstante, so bleibt das entsprechende Element der Variablenliste ungeändert.

Beispiele:

Datentyp	Wert	implizites Format	Ausgabe
FIXED(15)	127	F(6)	127
FLOAT(31)	3.28E+28	E(12,5,8)	3.28000E+28
BIT(8)	'EF'B4	B(8)	11101111

10.5.8 Das R-Format (R)

Manchmal werden gleiche Format-Positionslisten in mehr als einer Get- oder Put-Anweisung benutzt. Das R-Format dient dazu, diese Listen nur einmal zu beschreiben. Hierzu wird die Liste mit der sogenannten Format-Deklaration eingeführt.

Format-Deklaration ::=
    Bezeichner : **FORMAT** (Format-Position [, Format-Position] ···) ;

Beispiel:

Ftab : **FORMAT** (**X**(2), **F**(8,3), (3) (**X**(2), **E**(10,3))) ;

Ein solcherart vereinbartes Format kann in einer Get- oder Put-Anweisung unter Angabe seines Bezeichners benutzt werden:

R-Format ::=
    R (Bezeichner§Format)

Bei der Datenübertragung wird das R-Format durch die Format-Positionsliste ersetzt, die in der bezeichneten Format-Deklaration enthalten ist.

Beispiel:

PUT a, x, y, z **TO** Drucker **BY** **R**(Ftab) ;

Die Format-Positionsliste in der Format-Deklaration darf kein R-Format enthalten, das sich direkt oder indirekt (über eine weitere Format-Deklaration) auf die eigene Format-Deklaration bezieht.

10.6 Die Convert-Anweisung (CONVERT)

Die komfortable Umwandlung von Zahlenwerten in Zeichenketten und umgekehrt ist für viele Anwendungen sehr wichtig, z.B. für den Aufbau von Bildschirm-Masken oder zum Datenaustausch über Kommunikationsschnittstellen, die keine Binärdaten übertragen können. Die PUT- bzw. GET-Anweisung ermöglicht dies im Zusammenhang mit Datenstationen. In Anlehnung daran ist die CONVERT-Anweisung definiert, die formatierten Datenaustausch anstatt mit einer Datenstation mit einer Zeichenkette bzw. Zeichenkettenvariablen durchführt.

Die allgemeinen Formen der Convert-Anweisung sind:

Convert-To-Anweisung ::=
 CONVERT Ausdruck [, Ausdruck] ··· **TO** Name§Zeichenkette
 [**BY** Format-Position-Convert [, Format-Position-Convert] ···] ;

Convert-From-Anweisung ::=
 CONVERT Name§Variable [, Name§Variable] ··· **FROM** Ausdruck§Zeichenkette
 [**BY** Format-Position-Convert [, Format-Position-Convert] ···] ;

Format-Position-Convert ::=
 [Faktor] { Format **|** Position-Convert } **|**
 Faktor (Format-Position-Convert [, Format-Position-Convert] ···)

Position-Convert ::=
 RST (Name§Fehlervariable-FIXED) **|** **X** (Ausdruck) **|** **ADV** (Ausdruck) **|**
 POS (Ausdruck) **|** **SOP** (Name§Positionsvariable-FIXED)

Alle zugelassenen Formate haben die gleiche Bedeutung wie bei den PUT- und GET-Anweisungen. Die einzige Ausnahme ist das S-Format. Bei der Convert-Anweisung wird der Variablen im S-Format die Anzahl der Zeichen zugewiesen, die zu dem Zeitpunkt aus dem Ausdruck§Zeichenkette gelesen bzw. in die Name§Zeichenkette geschrieben wurde.

Beispiel:

```
DCL     (index, conv_error, number_of_bytes) FIXED ,
        wert                                 FLOAT ;

DCL     string_out      CHAR (40) ,
        string_in       CHAR (20) ;

...
CONVERT     'Index =', index  TO  string_out  BY  A , F (4), S (number_of_bytes) ;
CONVERT     index, wert  FROM  string_in  BY  RST (conv_error), F (4), E (10,2) ;
```

10.7 Die Take- und die Send-Anweisung

Die Take-Anweisung dient zur Eingabe, die Send-Anweisung zur Ausgabe von Daten. Diese Anweisungen sind für die Übertragung von Prozeßdaten vorgesehen, bzw. zum Datenaustausch mit Benutzer-spezifischen Treibern. Die Datenstation muß das Klassenattribut BASIC besitzen.

```
Take-Anweisung ::=
        TAKE  [  Name§Variable  ]  FROM  Name§Dation
                [  BY  RST-S-CTRL-Format  [ , RST-S-CTRL-Format ]...  ] ;

Send-Anweisung ::=
        SEND  [  Ausdruck  ]  TO  Name§Dation
                [  BY  RST-S-CTRL-Format  [ , RST-S-CTRL-Format ]...  ] ;

RST-S-CTRL-Format ::=
          RST  (  Name§Fehlervariable-FIXED  )
        | S  (  Name§Variable-FIXED  )
        | CONTROL  (  Ausdruck  [ , Ausdruck  [ , Ausdruck ] ]  )
```

Die Typen der "Variablen" in der Take-Anweisung bzw. des "Ausdrucks" in der Send-Anweisung sind implementationsabhängig.

Die Parameterangaben RST, S und CONTROL dürfen in beliebiger Reihenfolge, doch jeweils nur einmal angegeben werden. Die Bedeutungen der CONTROL- und S-Formate sind implementierungsabhängig und deshalb dem jeweiligen Benutzerhandbuch einer PEARL-Implementierung zu entnehmen.

Beispiel:

```
SYSTEM;
        Motor:    DIGEA * 1 * 1,4;

PROBLEM;
        SPC Motor DATION OUT BASIC;
        DCL Ein INV BIT(4) INIT ('1010'B1);

        SEND Ein TO Motor;
```

10.8 Fehlerbehandlung in E/A-Anweisungen (RST)

Normalerweise führen Fehler, die während der Abarbeitung von E/A-Anweisungen erkannt werden, zur Terminierung der verursachenden Task, und es wird eine Fehlermeldung ausgegeben. Diese Standard-Reaktion des PEARL-Systems wird unterdrückt durch die Angabe von

RST (Name§Fehlervariable-FIXED)

in der Parameterliste der OPEN-/CLOSE-Anweisung bzw. als Format- oder Positions-Element in den anderen Datenübertragungsanweisungen.

"Name" muß eine Variable vom Typ FIXED bezeichnen, in die im Fehlerfall eine Fehlernummer ungleich Null eingetragen wird. Bei fehlerfreier Ausführung der E/A-Anweisung wird die Variable auf Null gesetzt. Mögliche Fehler und ihre Identifizierung sind dem PEARL-Benutzerhandbuch des jeweiligen Rechnersystems zu entnehmen.

Die RST-Angabe kann an beliebiger Stelle in der Format- oder Positionsliste von PUT-, GET-, WRITE-, READ- und CONVERT-Anweisungen stehen; auch mehrfache Angaben mit unterschiedlichen Variablen sind erlaubt. Die RST-Angabe schaltet die Fehlerreaktion erst um, wenn das RST-Element in der Format-/Positionsliste ausgewertet wird. Eine E/A-Anweisung wird bei Erkennung eines Fehlers sofort abgebrochen und die zu diesem Zeitpunkt gültige Fehlerreaktion (Zuweisung einer Fehlernummer an eine RST-Variable oder die Systemreaktion) durchgeführt.

10.9 Interface für zusätzliche Treiber

Die Vielfalt der - insbesondere in der PC-Welt - existierenden E/A-Controller und Geräte erlaubt es dem Compiler-Lieferanten nicht, alle E/A-Geräte im Systemteil zu berücksichtigen und Systemnamen dafür einzurichten. Damit trotzdem spezielle E/A-Geräte von PEARL aus ansprechbar sind, kann für bestimmte Betriebssysteme ein Treiber-Interface vorgesehen werden, an das der PEARL-Programmierer selbst Treiber anschließen kann. Dieses Interface ist im jeweiligen Benutzerhandbuch (im Kapitel "Offene Treiber-Schnittstelle") beschrieben.

11. Signale

Bei der Ausführung bestimmter Anweisungen können interne Ereignisse, sogenannte Signale, eintreten, die zu einer Unterbrechung der laufenden Task führen; solche Signale sind beispielsweise ein Überlauf bei einer arithmetischen Operation, eine Division durch Null oder das Erreichen des Endes einer Datei („end-of-file“).

Ein Abbruch des Programms kann dadurch vermieden werden, daß bei Auftreten eines Signals eine entsprechende Fehlerbehandlung ermöglicht wird.

Die für ein Programm benötigten Signale werden im Systemteil erklärt, wobei ihnen frei wählbare Benutzernamen zugeordnet werden können. Zusätzlich kann eine Fehlerliste angegeben werden, mit der die Signal-Einplanung auf eine oder mehrere spezielle Fehlernummern eingeschränkt wird.

```
Benutzernamen-Vereinbarung§für-SIGNAL ::=
        Bezeichner§Benutzername  :   Bezeichner§SIGNAL-Systemname
                [ (  Bezeichner§Fehlernummer  [ , Bezeichner§Fehlernummer ] ...   ) ]
```

Die auf einem bestimmten Computer möglichen Signale und die zugeordneten Fehlernummern sind in dem jeweiligen Benutzerhandbuch unter Angabe ihrer Systemnamen und Bedeutung beschrieben.

Vor der Benutzung von Signalen müssen sie unter ihrem Benutzernamen im Problemteil auf Modulebene spezifiziert werden.

Beispiel:

IO_SIGNAL und ENDF sind Systemnamen; c_error_open ist eine vordefinierte Konstante.

```
MODULE ;
        SYSTEM ;
                OPEN_ERR   :  IO_SIGNAL (c_error_open) ;
                EOF        :  ENDF ;
                ...
        PROBLEM ;
                SPC   ( OPEN_ERR, EOF )   SIGNAL ;
                ...
MODEND ;
```

Die allgemeine Form der Spezifikation von Signalen lautet:

```
SIGNAL-Spezifikation ::=
        { SPECIFY  |  SPC }  Bezeichner-Angabe  SIGNAL  [ Global-Attribut ] ;
```

Die für den Eintritt eines Signals vorgesehene Reaktion wird mit der folgenden Anweisung eingeplant:

```
Einplanung-Signal-Reaktion ::=
        ON   Name§Signal   [ RST (  Name§Fehlervariable-FIXED  ) ]  :
                Signal-Reaktion

Signal-Reaktion ::=
        unmarkierte-Anweisung
```

Anstelle von unmarkierte-Anweisung sind alle Anweisungen außer der Anweisung Einplanung-Signal-Reaktion zugelassen, insbesondere also auch Blöcke oder Prozeduraufrufe.

Die Anweisung Einplanung-Signal-Reaktion ist innerhalb von BEGIN- und REPEAT-Blöcken, sowie als Signal-Reaktion nicht erlaubt.

- Gültigkeitsbereich der Einplanung einer Signal-Reaktion:

  Der Gültigkeitsbereich der Einplanung einer Signal-Reaktion erstreckt sich von der Ausführung der entsprechenden ON-Anweisung bis zum Ende der einplanenden Task bzw. Prozedur.

  Falls eine weitere Einplanung für dasselbe Signal (z.B. in einer von der einplanenden Task aufgerufenen Prozedur) durchlaufen wird, verdeckt diese die vorhergehende Einplanung bis zum Ende des Gültigkeitsbereichs der neuen Einplanung (im Beispiel bis zum Ende der einplanenden Prozedur).

- Verlassen einer Prozedur (Task) nach Ausführung einer Signal-Reaktion:

  Tritt innerhalb des Gültigkeitsbereichs der Einplanung einer Signal-Reaktion dieses Signal ein, so wird nach Ausführung der entsprechenden Signal-Reaktion die Prozedur (Task), in der die Signal-Reaktion eingeplant ist, durch ein implizites RETURN (TERMINATE) verlassen, wenn die Signal-Reaktion nicht durch eine GOTO-Anweisung verlassen wird.

- Gültigkeit von Signal-Einplanungen während der Ausführung einer Signal-Reaktion:

  Ist die Signal-Reaktion auf Prozedur-Ebene eingeplant, so sind während der Ausführung dieser Signal-Reaktion nur noch die ON-Einplanungen gültig, die bereits vor dem Aufruf dieser Prozedur ausgeführt wurden.

  Ist die Signal-Reaktion auf Task-Ebene eingeplant, sind während der Ausführung dieser Signal-Reaktion keine ON-Einplanungen mehr gültig.

- Gültigkeit von Signal-Einplanungen nach Verlassen einer Signal-Reaktion durch GOTO:

  Wird die Bearbeitung einer auf Prozedur- bzw. Task-Ebene eingeplanten Signal-Reaktion mit GOTO verlassen, sind danach die bereits vor der Auslösung des Signals ausgeführten Signal-Einplanungen, insbesondere die des ausgelösten Signals, wieder gültig.

Zum Test der für ein Signal eingeplanten Reaktion kann der Eintritt eines Signals analog zum Eintritt eines Interrupts simuliert werden:

```
Induce-Anweisung ::=
        INDUCE  Name§Signal  [ RST (  Ausdruck§Fehlernummer  ) ] ;
```

Während durch Interrupts auf asynchrone Ausnahmesituationen (d.h. von außen einwirkende Einflüsse) reagiert werden kann, dient die Signalbehandlung ausschließlich zur Reaktion auf synchrone Fehlerzustände (d.h. Ursache und Behandlung des Fehlerzustandes gehen auf dieselbe Task zurück).

Beispiel:

Die Prozedur „Auswertung“ soll ein im Laufe eines Tages erstelltes Logbuch sequentiell auswerten; die einzelnen Datenelemente des Logbuchs sind vom Typ Ereignis.

```
. . .
PROBLEM ;
        SPC     EOF  SIGNAL ,
                Band  DATION  INOUT  ALL ;
        TYPE    Ereignis  . . . ;
        DCL     Logbuch  DATION  IN Ereignis  DIM ( * )  FORWARD  CREATED ( Band ) ;

Auswertung :    PROC ;
        DCL     Eingabe  Ereignis ;
        . . .
        OPEN  Logbuch ;
        ON  EOF :
                BEGIN
                        CLOSE  Logbuch ;
                END ; !ON EOF
        . . .
        REPEAT
                READ  Eingabe  FROM  Logbuch ;
                . . .
        END ;
        END ; ! Auswertung
```

Zum Testen könnte man sporadisch anstelle der Read-Anweisung die Anweisung

```
INDUCE  EOF ;
```

ausführen.

Wenn ein Signal ausgelöst wird, sei es durch einen Fehlerzustand oder durch eine Induce-Anweisung, und keine Signal-Reaktion dafür eingeplant ist, wird die

Systemreaktion ausgelöst (i.a. also Fehlermeldung des Laufzeitsystems und Abbruch der verursachenden Task).

Durch die Angabe einer Variablen nach RST in der Signal-Einplanung erhält der Programmierer Zugriff auf die Fehlernummer (Fehlerursache). In diesem Fall kann durch

INDUCE Signalname **RST** (Fehlernummer) ;

das Signal „Signalname“ für den Fehler mit der Nummer „Fehlernummer“ ausgelöst werden.

Beispiel:

Das Signal „TaskSignal“ soll mit der Fehlernummer „1010“ simuliert werden und die verursachende Task mit der Ausgabe der Fehlernummer auf der Konsole reagieren:

```
PROBLEM ;
        SPC     TaskSignal SIGNAL ;
        DCL     FehlerNummer FIXED ;

Regler: TASK PRIO 20 ;
        ON  TaskSignal RST (FehlerNummer) :
            PUT FehlerNummer TO Konsole ;
            . . .
        Test;
            . . .
        END ; ! Regler

Start:  TASK MAIN ;
        ALL Ta ACTIVATE Regler ;
        END ; ! Start

Test:   PROC ;
        INDUCE TaskSignal RST (1010) ;
        . . .
        END; ! Test
```

Anhang A

A.1 Datentypen und ihre Verwendbarkeit

Die folgende Übersicht zeigt für jeden der zur Verfügung stehenden Datentypen, ob Objekte dieses Typs

- zu Feldern zusammengefaßt werden
- als Strukturkomponente auftreten
- formaler Prozedurparameter sein
- Ergebnis einer Funktionsprozedur sein
- Wert einer Referenz-Variablen sein
- von und zu Datenstationen übertragen werden
- mit Zuweisungsschutz versehen werden
- global sein oder
- mit dem Initialisierungsattribut versehen werden

dürfen.

Typ	**Verwendung**								
	Feld	Struk- tur	Para- meter	Resultat- Typ	Ref- Wert	Dation- Klasse	INV	GLO- BAL	INIT
FIXED	x	x	x	x	x	x	x	x	x
FLOAT	x	x	x	x	x	x	x	x	x
BIT	x	x	x	x	x	x	x	x	x
CHAR	x	x	x	x	x	x	x	x	x
CLOCK	x	x	x	x	x	x	x	x	x
DUR	x	x	x	x	x	x	x	x	x
SEMA	x	-	x	-	x	-	-	x	-
BOLT	x	-	x	-	x	-	-	x	-
IRPT	x	-	x	-	x	-	-	x	-
SIGNAL	x	-	x	-	x	-	-	x	-
DATION	x	-	x	-	x	-	-	x	-
Feld	-	x	x	-	x	-	x	x	x
STRUCT	x	x	x	x	x	x	x	x	x
neuer Typ	x	x	x	x	x	x	x	x	x
REF	x	x	x	x	-	-	x	x	x
Prozedur	-	-	-	-	x	-	-	x	-
TASK	-	-	-	-	x	-	-	x	-
FORMAT	-	-	-	-	-	-	-	-	-
REF CHAR ()	x	x	x	-	-	-	x	x	x
REF PROC	x	x	x	-	-	-	x	x	x
REF TASK	x	x	x	x	-	-	x	x	x
REF STRUCT []	x	x	x	x	-	-	x	x	x

Objekte vom Typ SEMA, BOLT, IRPT, SIGNAL, DATION oder Feld dürfen nur mittels Identifizierung (IDENT) als Prozedur-Parameter übergeben werden.

A.2 Vordefinierte Funktionen

Dieser Anhang beschreibt die dem PEARL-Compiler bekannten Funktionen. Sie können in den einzelnen Modulen benutzt werden, ohne sie vorher zu spezifizieren. Falls eine der Funktionen in einem Modul spezifiziert wird, darf auf Modul-Ebene kein Objekt mit deren Namen existieren.

A.2.1 Die Funktion NOW

Die Funktionsprozedur NOW liefert die aktuelle Uhrzeit, bzw. Systemzeit als einen Wert vom Typ CLOCK zurück. Eine Spezifikation der Funktion sieht folgendermaßen aus:

```
SPC NOW PROC RETURNS ( CLOCK ) GLOBAL ;
```

A.2.2 Die Funktion DATE

Das aktuelle Datum erhält man durch einen Aufruf der Funktionsprozedur DATE. Das Funktionsergebnis ist eine Zeichenkettenkonstante der Länge 10, die das Datum in der Form "Jahr-Monat-Tag" enthält. Hier ein Beispiel für den 5. Dezember 1991: "1991-12-05". Die Funktion kann so spezifiziert werden:

```
SPC DATE PROC RETURNS ( CHAR( 10 ) ) GLOBAL ;
```

A.3 Syntax

In der vorliegenden Syntaxbeschreibung werden folgende Meta-Zeichen benutzt:

Meta-Zeichen	Bedeutung
::=	Einführung eines Namens (Nichtterminal-Symbol) für eine Sprachform
[ ]	Klammerung optionaler Teile einer Sprachform
|	Trennung von alternativen Teilen einer Sprachform
{ }	Zusammenfassung mehrerer Elemente zu einem neuen Element
...	ein oder mehrfache Wiederholung des vorangehenden Elementes (oder mehrerer durch { } oder [ ] geklammerte Elemente)
§	trennt einen erläuternden Kommentar von einem Sprachform-Namen
/* */	Kommentarklammern: schließt einen erläuternden Text ein, der evtl. anstelle einer formalen Beschreibung die Sprachform genauer erklärt.

Alle anderen in den Syntaxregeln vorkommenden Elemente sind entweder Namen von Sprachformen oder Terminal-Symbole. Beispiele für Terminal-Symbole sind die PEARL-Schlüsselworte (fett gedruckt) oder die Zeichen Strichpunkt ";", Klammer-auf "(" und Klammer-zu ")" oder das Apostroph "'"; die Terminal-Symbole eckige-Klammer-auf "**[**" und eckige-Klammer-zu "**]**" werden fett gedruckt, um sie von den Meta-Symbolen für optionale Teile unterscheiden zu können. Achtung: die runden Klammern sind keine Meta-Zeichen und haben deshalb auch keine gruppierende Wirkung!

Das Symbol PEARL-Programm ist das Startsymbol der folgenden Syntaxbeschreibung.

A.3.1 Programm

PEARL-Programm ::=
    Modul...

Modul ::=
    MODULE [[( ] Bezeichner§des-Moduls [ )]] ;
    { Systemteil [Problemteil] | Problemteil }
    MODEND ;

A.3.2 Systemteil

Systemteil ::=
  SYSTEM ;
    [Benutzernamen-Vereinbarung§für-Dation-Interrupt-oder-Signal] ···

Benutzernamen-Vereinbarung ::=
  Bezeichner§Benutzername : Bezeichner§Systemname [(nngz§Index)]
    [* nngz§Kanal [* nngz§Position [, nngz§Breite]]] ;
  | Bezeichner§Benutzername : Bezeichner§SIGNAL-Systemname
    [(Bezeichner§Fehlernummer [, Bezeichner§Fehlernummer] ···)] ;

nngz ::=
  ganze-Zahl-ohne-Genauigkeit§nicht-negativ

A.3.3 Grundelemente

Ziffer ::=
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Buchstabe ::=
  A | B | C | . . . | Z | a | b | c | . . . | z

Bezeichner ::=
  Buchstabe [Buchstabe | Ziffer | _] ···

Konstante ::=
  ganze-Zahl | Gleitpunktzahl
  | Bitkettenkonstante | Zeichenkettenkonstante
  | Uhrzeitkonstante | Dauerkonstante
  | **NIL**

ganze-Zahl ::=
  ganze-Zahl-ohne-Genauigkeit [(Genauigkeit)]

ganze-Zahl-ohne-Genauigkeit ::=
  Ziffer ··· | { 0 | 1 } ··· B

Genauigkeit ::=
  ganze-Zahl-ohne-Genauigkeit

Gleitpunktzahl ::=
  Gleitpunktzahl-ohne-Genauigkeit [(Genauigkeit)]

Gleitpunktzahl-ohne-Genauigkeit ::=
  { Ziffer ··· **.** [Ziffer ···] | **.** Ziffer ··· } [Exponent]
  | Ziffer ··· Exponent

Exponent ::=
  E [+ | -] Ziffer ···

Bitkettenkonstante ::=
' B1-Ziffer··· ' { B | B1 } | ' B2-Ziffer··· ' B2 | ' B3-Ziffer··· ' B3 | ' B4-Ziffer··· ' B4

B1-Ziffer ::= 0 | 1

B2-Ziffer ::= 0 | 1 | 2 | 3

B3-Ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

B4-Ziffer ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F

Zeichenkettenkonstante ::=
' { Zeichen-außer-Apostroph | '' | Kontrollzeichen-Sequenz } ··· '

Zeichen-außer-Apostroph ::=
Ziffer | Buchstabe | | _ | + | - | * | / | \ | (|) | [|] | : | . | ; | , | = | < | > | !
| /* weitere druckbare Zeichen des Maschinenzeichensatzes */

Kontrollzeichen-Sequenz ::=
'\ { B4-Ziffer B4-Ziffer } ··· \'

Uhrzeitkonstante ::=
Ziffer ··· : Ziffer ··· : Ziffer ··· [. Ziffer ···]

Dauerkonstante ::=
Stundenangabe [Minutenangabe] [Sekundenangabe]
| Minutenangabe [Sekundenangabe]
| Sekundenangabe

Stundenangabe ::=
ganze-Zahl-ohne-Genauigkeit **HRS**

Minutenangabe ::=
ganze-Zahl-ohne-Genauigkeit **MIN**

Sekundenangabe ::=
{ ganze-Zahl-ohne-Genauigkeit | Gleitpunktzahl-ohne-Genauigkeit } **SEC**

konstanter-Ausdruck ::=
{ + | - } Gleitpunktzahl
| { + | - } Dauerkonstante
| konstanter-FIXED-Ausdruck

konstanter-FIXED-Ausdruck ::=
Term [{ + | - } Term] ···

Term ::=
Faktor [{ ***** | // | **REM** } Faktor] ···

Faktor ::=
[+ | -]
{ ganze-Zahl
| (konstanter-FIXED-Ausdruck)
| **TOFIXED** { Zeichenkettenkonstante§der-Länge-1 |
Bitkettenkonstante }
| Bezeichner§benannte-FIXED-Konstante
}
[**FIT** konstanter-FIXED-Ausdruck]

A.3.4 Problemteil

Problemteil ::=
PROBLEM ; [Vereinbarung ···]

Vereinbarung ::=
Deklaration | Spezifikation | Identifikation

A.3.4.1 Deklaration

Deklaration ::=
Längen-Vereinbarung
| Typ-Vereinbarung
| Variablen-Deklaration
| Format-Deklaration
| Prozedur-Deklaration
| Task-Deklaration
| Operator-Vereinbarung
| Rang-Vereinbarung

Längen-Vereinbarung ::=
LENGTH { **FIXED** | **FLOAT** | **BIT** | **CHARACTER** | **CHAR** } (
Genauigkeit) ;

Typ-Vereinbarung ::=
TYPE Bezeichner§für-Typ { einfacher-Typ | Typ-Struktur } ;

Variablen-Deklaration ::=
{ **DECLARE** | **DCL** } Deklarations-Satz [, Deklarations-Satz] ··· ;

Deklarations-Satz ::=
Bezeichner-Angabe [Dimensions-Attribut]
{ Problemdaten-Attribut | Sema-Attribut
| Bolt-Attribut | Dation-Attribut }

Bezeichner-Angabe ::=
Bezeichner | (Bezeichner [, Bezeichner] ···)

Dimensions-Attribut ::=
(Dimensionsgrenzen [, Dimensionsgrenzen] ···)

Dimensionsgrenzen ::=
[konstanter-FIXED-Ausdruck§für-untere-Grenze :]
konstanter-FIXED-Ausdruck§für-obere-Grenze

Problemdaten-Attribut ::=
[**INV**] { einfacher-Typ | strukturierter-Typ | Typ-Referenz }
[Resident-Attribut] [Global-Attribut] [Initialisierungsattribut]

einfacher-Typ ::=
{ **FIXED** | **FLOAT** | **BIT** | **CHAR** | **CHARACTER** }
[(konstanter-FIXED-Ausdruck§Genauigkeit-bzw.-Länge)]
| **CLOCK**
| { **DUR** | **DURATION** }

strukturierter-Typ ::=
Typ-Struktur | Bezeichner§für-neu-vereinbarten-Typ

Typ-Struktur ::=
STRUCT **[** Komponenten-Deklaration
[, Komponenten-Deklaration] ··· **]**

Komponenten-Deklaration ::=
Bezeichner-Angabe§für-Strukturkomponente [Dimensions-Attribut]
Typ-Attribut-in-Struktur-Vereinbarung

Typ-Attribut-in-Struktur-Vereinbarung ::=
[**INV**] { einfacher-Typ | strukturierter-Typ | Typ-Referenz }

Typ-Referenz ::=
REF
{ [virtuelle-Dimensionsliste] [**INV**] { einfacher-Typ
| strukturierter-Typ }
| [virtuelle-Dimensionsliste] { Typ-Dation | **SEMA** | **BOLT** }
| Typ-Prozedur | **TASK** | **INTERRUPT** | **IRPT** | **SIGNAL**
| Typ-VOID | **CHAR**( )
}

virtuelle-Dimensionsliste ::=
([, ···])

Typ-VOID ::=
STRUCT [ ] /* nur in Verbindung mit REF erlaubt */

Sema-Attribut ::=
SEMA [Resident-Attribut] [Global-Attribut]
[**PRESET** (konstanter-FIXED-Ausdruck
[, konstanter-FIXED-Ausdruck] ···)]

Bolt-Attribut ::=
BOLT [Resident-Attribut] [Global-Attribut]

Resident-Attribut ::=
RESIDENT

Global-Attribut ::=
GLOBAL [(Bezeichner§Modul)]

Initialisierungsattribut ::=
{ **INITIAL** | **INIT** } (Init-Element [, Init-Element] ···)

Init-Element ::=
Konstante
| Bezeichner§benannte-Konstante
| konstanter-Ausdruck

Dation-Attribut ::=
Typ-Dation
[Resident-Attribut] [Global-Attribut]
CREATED (Bezeichner§Benutzername-für-Sytemdation)

Typ-Dation ::=
DATION Quelle-Senke-Attribut Klassenattribut
[Topologie] [Zugriffsattribut] [Kontroll-Attribut]

Quelle-Senke-Attribut ::=
IN | **OUT** | **INOUT**

Klassenattribut ::=
ALPHIC | **BASIC** | Typ-der-Übertragungsdaten

Typ-der-Übertragungsdaten ::=
ALL | einfacher-Typ | Bezeichner§für-neu-vereinbarten-Typ
| Typ-Struktur
/* eine Struktur als Transfer-Typ darf keine Referenz-Variablen enthalten */

Topologie ::=
DIM ({ * | konstanter-FIXED-Ausdruck }
[, { * | konstanter-FIXED-Ausdruck }
[, { * | konstanter-FIXED-Ausdruck }]])
[**TFU** [**MAX**]]
| **DIM** ([, [,]]) /* virtuelle Dimensionsangabe nur bei SPC erlaubt */

Zugriffsattribut ::
{ **DIRECT** | **FORWARD** | **FORBACK** }
[**NOCYCL** | **CYCLIC**]
[**STREAM** | **NOSTREAM**]

Kontroll-Attribut ::=
CONTROL (**ALL**)

Format-Deklaration ::=
Bezeichner : **FORMAT** (Format-oder-Position
[, Format-oder-Position] ···) ;

Prozedur-Deklaration ::
Bezeichner : { **PROC** | **PROCEDURE** } [Liste-formaler-Parameter]
[Resultat-Attribut]
[Resident-Attribut] [Reentrant-Attribut] [Global-Attribut] ;

Prozedurkörper

END ;

Prozedurkörper ::=
[Vereinbarung ···] [Anweisung ···]

Liste-formaler-Parameter ::=
(formaler-Parameter [, formaler-Parameter] ···)

formaler-Parameter ::=
Bezeichner-Angabe [virtuelle-Dimensionsliste]
Parameter-Typ [**IDENT** | **IDENTICAL**]

Parameter-Typ ::=
[**INV**] { einfacher-Typ | strukturierter-Typ | Typ-Referenz } |
Typ-Echtzeit-Objekt | Typ-Dation

Typ-Echtzeit-Objekt ::=
SEMA | **BOLT** | { **INTERRUPT** | **IRPT** } | **SIGNAL**

Resultat-Attribut ::=
RETURNS (Resultat-Typ)

Resultat-Typ ::=
einfacher-Typ | strukturierter-Typ | Typ-Referenz

Reentrant-Attribut ::=
REENT

Task-Deklaration ::=
Bezeichner : **TASK** [Prioritätsangabe] [**MAIN**]
[Resident-Attribut] [Global-Attribut] ;

Prozedurkörper

END ;

Prioritätsangabe ::=
{ **PRIORITY** | **PRIO** } konstanter-FIXED-Ausdruck§größer-Null

Operator-Vereinbarung ::=
OPERATOR Op-Name (Op-Parameter [, Op-Parameter])
Resultat-Attribut ;

Prozedurkörper

END ;

Op-Name ::=
Bezeichner | + | - | * | / | // | **
| == | /= | <= | >= | < | > | <> | ><

Op-Parameter ::=
Bezeichner [virtuelle-Dimensionsliste] Parameter-Typ
[**IDENT** | **IDENTICAL**]

Rang-Vereinbarung ::=
PRECEDENCE Op-Name ({ 1 | 2 | 3 | 4 | 5 | 6 | 7 }) ;

A.3.4.2 Spezifikation und Identifikation

Spezifikation ::=
  { **SPC | SPECIFY** } Spezifikations-Angabe
  [, Spezifikations-Angabe] ··· ;

Spezifikations-Angabe ::=
  Bezeichner-Angabe
  { Spezifikationsattribut **|** Prozedur-Benutzungsattribut
  | Task-Benutzungsattribut }

Spezifikationsattribut ::=
  [virtuelle-Dimensionsliste]
  { [**INV**] { einfacher-Typ **|** strukturierter-Typ **|** Typ-Referenz }
  | SEMA | BOLT | INTERRUPT | IRPT | SIGNAL | Typ-Dation
  }
  [Resident-Attribut] [Global-Attribut]

Prozedur-Benutzungsattribut ::=
  { **ENTRY |** [:] **PROC** } [Parameter-Liste-für-SPC]
  [Resultat-Attribut]
  [Resident-Attribut] [Reentrant-Attribut] Global-Attribut

Parameter-Liste-für-SPC ::=
  (formaler-Parameter-für-SPC [, formaler-Parameter-für-SPC] ···)

formaler-Parameter-für-SPC ::=
  [Bezeichner§nur-zur-Dokumentation] [virtuelle-Dimensionsliste]
  Parameter-Typ [**IDENTICAL | IDENT**]

Typ-Prozedur ::=
  PROC [Parameter-Liste-für-SPC] [Resultat-Attribut]
  /* für REF PROC */

Task-Benutzungsattribut ::=
  TASK [Resident-Attribut] Global-Attribut

Identifikation ::=
  { **SPC | SPECIFY** } Bezeichner [**INV**] Typ Identifikationsattribut ;

Identifikationsattribut ::=
  IDENT (Name§Objekt)

A.3.4.3 Ausdrücke

Ausdruck ::=
[monadischer-Operator] ··· Operand
[dyadischer-Operator Ausdruck]

monadischer-Operator ::=
+ | - | Bezeichner§monadischer-Operator

dyadischer-Operator ::=
+ | - | * | / | // | ** | < | > | <= | >= | == | /= | >< | <> |
Bezeichner§dyadischer-Operator

Operand ::=
Konstante | Name | Funktionsaufruf | bedingter-Ausdruck
| Dereferenzierung | Kettenausschnitt | (Ausdruck) | (Zuweisung)
| **PRIO** [(Name§Task)] | **TASK** [(Name§Task)]
| **TRY** Name§Sema

Name ::=
Bezeichner [(Index [, Index] ···)] [**.** Name]

Index ::=
Ausdruck§mit-ganzer-Zahl-als-Wert

Funktionsaufruf ::=
Bezeichner§Funktionsprozedur [Liste-aktueller-Parameter]

Liste-aktueller-Parameter ::=
(Ausdruck [, Ausdruck] ···)

bedingter-Ausdruck ::=
IF Ausdruck§vom-Typ-BIT(1) **THEN** Ausdruck **ELSE** Ausdruck **FIN**

Dereferenzierung ::=
CONT { Name§Referenz | Funktionsaufruf }

Kettenausschnitt ::=
Name§Kette **.** { **BIT** | **CHAR** | **CHARACTER** }
({ konstanter-FIXED-Ausdruck [**:** konstanter-FIXED-Ausdruck]
| Ausdruck [**:** Ausdruck + konstanter-FIXED-Ausdruck]
| Ausdruck [**:** Ausdruck] })

A.3.4.4 Anweisungen

Anweisung ::=
    [Bezeichner§Sprungziel **:**] ··· unmarkierte-Anweisung

unmarkierte-Anweisung ::=
    Leeranweisung | Zuweisung | Block
    | sequentielle-Steueranweisung
    | Echtzeit-Anweisung | Convert-Anweisung | EA-Anweisung

Leeranweisung ::=
    ;

Zuweisung ::=
    { { Name§Variable
    | Dereferenzierung
    | Kettenausschnitt
    | Name§Struktur } { := | = } } ··· Ausdruck ;

Block ::=
    BEGIN
        [Vereinbarung ···]
        [Anweisung ···]
    END [Bezeichner§Block] ;

sequentielle-Steueranweisung ::=
    bedingte-Anweisung | Anweisungsauswahl | Wiederholung
    | Exit-Anweisung
    | Prozedur-Aufruf | Return-Anweisung | Sprung-Anweisung

bedingte-Anweisung ::=
    IF Ausdruck§vom-Typ-BIT(1)
        THEN [Anweisung ···]
    [**ELSE** [Anweisung ···]]
    FIN ;

Anweisungsauswahl ::=
    Anweisungsauswahl-1 | Anweisungsauswahl-2

Anweisungsauswahl-1 ::=
    CASE Ausdruck§mit-ganzer-Zahl-als-Wert
    { **ALT** [Anweisung ···] } ···
    [**OUT** [Anweisung ···]]
    FIN ;

```
Anweisungsauswahl-2 ::=
        CASE   Case-Index
          {   ALT   (  Case-Liste  )  [ Anweisung ··· ]   } ···
          [   OUT                     [ Anweisung ··· ]   ]
        FIN  ;

Case-Index ::=
        Ausdruck§mit-Wert-vom-Typ-FIXED-oder-CHAR(1)

Case-Liste ::=
        Index-Bereich   [  ,  Index-Bereich  ] ···

Index-Bereich ::=
           konstanter-FIXED-Ausdruck   [  :  konstanter-FIXED-Ausdruck  ]
        |  Zeichenkettenkonstante§der-Länge-1
           [  :  Zeichenkettenkonstante§der-Länge-1  ]

Wiederholung ::=
        [ FOR     Bezeichner§Laufvariable ]
        [ FROM    Ausdruck§Anfangswert ]
        [ BY      Ausdruck§Schrittweite ]
        [ TO      Ausdruck§Endwert ]
        [ WHILE   Ausdruck§Bedingung ]
        REPEAT
                [ Vereinbarung ··· ]
                [ Anweisung ··· ]
        END  [ Bezeichner§Schleife ]  ;

Exit-Anweisung ::=
        EXIT  [  Bezeichner§Block-oder-Schleife  ]  ;

Prozedur-Aufruf ::=
        [ CALL ]   Name§Unterprogramm-Prozedur
                        [ Liste-aktueller-Parameter ]  ;

Return-Anweisung ::=
        RETURN  [  (  Ausdruck  )  ]  ;

Sprung-Anweisung ::=
        GOTO  Bezeichner§Marke  ;


Echtzeit-Anweisung ::=
           Task-Steueranweisung  |   Task-Koordinierungsanweisung
        |  Interrupt-Anweisung       |   Signal-Anweisung

Task-Steueranweisung ::=
           Task-Starten       |  Task-Beenden
        |  Task-Anhalten      |  Task-Fortsetzen
        |  Task-Verzögern     |  Task-Ausplanen
```

Task-Starten ::=
[Startbedingung] **ACTIVATE** Name§Task [Prioritäts-Ausdruck] ;

Prioritäts-Ausdruck ::=
{ **PRIO | PRIORITY** } Ausdruck§positive-ganze-Zahl

Startbedingung ::=
AT Ausdruck§Uhrzeit [Frequenz]
| **AFTER** Ausdruck§Dauer [Frequenz]
| **WHEN** Name§Interrupt [**AFTER** Ausdruck§Dauer] [Frequenz]
| Frequenz

Frequenz ::=
{ **ALL | EVERY** } Ausdruck§Dauer
[{ **UNTIL** Ausdruck§Uhrzeit } **|** { **DURING** Ausdruck§Dauer }]

Task-Beenden ::=
TERMINATE [Name§Task] ;

Task-Anhalten ::=
SUSPEND [Name§Task] ;

Task-Fortsetzen ::=
[**AT** Ausdruck§Uhrzeit **|** **AFTER** Ausdruck§Dauer **|** **WHEN**
Name§Interrupt]
CONTINUE [Name§Task] [Prioritätsangabe] ;

Task-Verzögern ::=
{ **AT** Ausdruck§Uhrzeit **|** **AFTER** Ausdruck§Dauer **|** **WHEN**
Name§Interrupt }
RESUME ;

Task-Ausplanen ::=
PREVENT [Name§Task] ;

Task-Koordinierungsanweisung ::=
{ **REQUEST | RELEASE** } Name§Sema
[, Name§Sema] ··· ;
| { **RESERVE | FREE | ENTER | LEAVE** } Name§Bolt
[, Name§Bolt] ··· ;

Interrupt-Anweisung ::=
{ **ENABLE | DISABLE | TRIGGER** } Name§Interrupt ;

Signal-Anweisung ::=
ON Name§Signal [RST (Name§Fehlervariable-FIXED)] **:**
unmarkierte-Anweisung
| **INDUCE** Name§Signal [RST (Ausdruck§Fehlernummer-FIXED)] ;

Convert-Anweisung ::=
Convert-To-Anweisung | Convert-From-Anweisung

Convert-To-Anweisung ::=
CONVERT Ausdruck [, Ausdruck] ··· **TO** Name§Zeichenketten-Variable
[**BY** Format-oder-Position-Convert [, Format-oder-Position-Convert] ···] ;

Convert-From-Anweisung ::=
CONVERT Name§Variable [, Name§Variable] ··· **FROM** Ausdruck§Zeichenkette
[**BY** Format-oder-Position-Convert [, Format-oder-Position-Convert] ···] ;

Format-oder-Position-Convert ::=
[Faktor] { Format | Position-Convert }
| Faktor (Format-oder-Position-Convert
[, Format-oder-Position-Convert] ···)

Position-Convert ::=
RST (Name§Fehlervariable-FIXED)
| **X** [(Ausdruck)]
| { **POS | ADV** } (Ausdruck)
| **SOP** (Name§Positionsvariable-FIXED)

EA-Anweisung ::=
Open-Anweisung | Close-Anweisung
| Put-Anweisung | Get-Anweisung
| Write-Anweisung | Read-Anweisung
| Send-Anweisung | Take-Anweisung

Open-Anweisung ::=
OPEN Name§Dation [**BY** Open-Parameter
[, Open-Parameter] ···] ;

Open-Parameter ::=
RST (Name§Fehlervariable-FIXED)
| **IDF** ({ Name§Zeichenketten-Variable | Zeichenkettenkonstante })
| { **OLD** | **NEW** | **ANY** }
| { **CAN** | **PRM** }

Close-Anweisung ::=
CLOSE Name§Dation [**BY** Close-Parameter
[, Close-Parameter] ···] ;

Close-Parameter ::=
RST (Name§Fehlervariable-FIXED)
| { **CAN** | **PRM** }

Put-Anweisung ::=
PUT [{ Ausdruck | Ausschnitt } [, { Ausdruck | Ausschnitt }] ⋯]
TO Name§Dation [**BY** Format-oder-Position
[, Format-oder-Position] ⋯] ;

Get-Anweisung ::=
GET [{ Name | Ausschnitt } [, { Name | Ausschnitt }] ⋯]
FROM Name§Dation [**BY** Format-oder-Position
[, Format-oder-Position] ⋯] ;

Write-Anweisung ::=
WRITE [{ Ausdruck | Ausschnitt } [, { Ausdruck | Ausschnitt }] ⋯]
TO Name§Dation [**BY** Position [, Position] ⋯] ;

Read-Anweisung ::=
READ [{ Name | Ausschnitt } [, { Name | Ausschnitt }] ⋯]
FROM Name§Dation [**BY** Position [, Position] ⋯] ;

Send-Anweisung ::=
SEND [Ausdruck] **TO** Name§Dation
[**BY** RST-S-CTRL-Format [, RST-S-CTRL-Format] ⋯] ;

Take-Anweisung ::=
TAKE [Name] **FROM** Name§Dation
[**BY** RST-S-CTRL-Format [, RST-S-CTRL-Format] ⋯] ;

RST-S-CTRL-Format ::=
RST (Name§Fehlervariable-FIXED)
| **S** (Name§Variable-FIXED)
| **CONTROL** (Ausdruck [, Ausdruck [, Ausdruck]])

Ausschnitt ::=
Name§Feld ([Index ,] ⋯ Index **:** Index)
/* bei einer indizierten Feldansprache darf in der letzten Dimension ein Bereich angegeben werden: "Ausdruck : Ausdruck"; die Elemente dieses eindimensionalen Slices werden in aufsteigender Reihenfolge von der EA verarbeitet. */

Format-oder-Position ::=
[Faktor] { Format | Position }
| Faktor (Format-oder-Position [, Format-oder-Position] ⋯)

Faktor ::=
(Ausdruck§ganze-Zahl-größer-Null)
| ganze-Zahl-ohne-Genauigkeit§größer-Null

```
Format ::=
        { F | E } (  Ausdruck  [ , Ausdruck  [ , Ausdruck ] ]  )
      | { B | B1 | B2 | B3 | B4 | A }  [ (  Ausdruck  ) ]
      | { T | D } (  Ausdruck  [ , Ausdruck ]  )
      | LIST
      | R  (  Bezeichner§Format  )
      | S  (  Name§Längenvariable-FIXED  )

Position ::=
        RST  (  Name§Fehlervariable-FIXED  )
      | { X | SKIP | PAGE }  [ (  Ausdruck  ) ]
      | { POS | ADV } (  Ausdruck  [ , Ausdruck  [ , Ausdruck ] ]  )
      | { COL | LINE } (  Ausdruck  )
      | SOP  (  Name  [ , Name  [ , Name ] ]  /* Positionsvariablen-FIXED */  )
```

A.4 Liste der Schlüsselwörter mit Kurzformen

Die Angabe hinter einem Schlüsselwort verweist auf den Paragraphen, in dem es eingeführt wird.

ACTIVATE 9.2.2
AFTER 9.2.1
ALL 9.2.1
ALPHIC 10.2
ALT 7.2
AT 9.2.1

BASIC 10.2
BEGIN 4.4
BIT 5.4
BOLT 9.3.2
BY 7.3, 10.3

CALL 8.2
CASE 7.2
CHARACTER , **CHAR** 5.5
CLOCK 5.7
CLOSE 10.3
CONT 5.9
CONTINUE 9.2.5
CONTROL 10.2
CONVERT 10.2
CREATE (Full PEARL)
CREATED 10.2
CYCLIC 10.2

DATION 10.2
DECLARE , **DCL** 4.1
DELETE (Full PEARL)
DIM 10.2
DIRECT 10.2
DISABLE 9.4.2
DURATION , **DUR** 5.8
DURING 9.2.1

ELSE 7.1
ENABLE 9.4.2
END 4.4, 8.1, 9.1
ENTER 9.3.2
ENTRY 8.1
EVERY 9.2.1
EXIT 7.5

FIN 7.1
FIXED 5.2
FLOAT 5.3
FOR 7.3
FORBACK 10.2
FORMAT 10.5.8
FORWARD 10.2
FREE 9.3.2
FROM 7.3, 10.4

GET 10.5
GLOBAL 4.1
GOTO 7.4

HRS 5.8

IDENTICAL , IDENT 4.1, 8.1
IF 7.1
IN 10.2
INDUCE 11
INITIAL , **INIT** 5.14
INLINE (Full PEARL)

A.5 Sonstige Wortsymbole in PEARL

Die Angabe hinter dem Wortsymbol verweist auf den Paragraphen, in dem es eingeführt wird.

A 10.5.2
ABS 6.1.1
ADV 10.4
AND 6.1.2
ANY 10.3

B 10.5.4
B1 10.5.4
B2 10.5.4
B3 10.5.4
B4 10.5.4

CAN 10.3
CAT 6.1.2
COL 10.4
COS 6.1.1
CSHIFT 6.1.2

D 10.5.6
DATE A2.2

E 10.5.2
ENTIER 6.1.1
EQ 6.1.2
EXOR 6.1.2
EXP 6.1.1

F 10.5.1
FIT 6.1.2

GE 6.1.2
GT 6.1.2

IDF 10.3

LE 6.1.2
LINE 10.4
LIST 10.5.7
LN 6.1.1
LT 6.1.2
LWB 6.1.1, 6.1.2

NE 6.1.2
NEW 10.3
NOT 6.1.1
NOW A2.1

OLD 10.3
OR 6.1.2

PAGE 10.4
POS 10.4
PRM 10.3

R 10.5.8
REM 6.1.2
ROUND 6.1.1
RST 10.3, 10.8

S 10.5.3
SIGN 6.1.1
SIN 6.1.1
SKIP 10.4
SOP 10.4
SQRT 6.1.1